Sorbonne Université - École Doctorale EDITE de Paris (ED130)

Université de Biskra - Departement d'informatique - Laboratoire d'informatique intelligente (LINFI)

THÈSE EN COTUTELLE

pour l'obtention des diplômes de

Doctorat en Informatique de Sorbonne Université
Doctorat en Informatique de l'Université de Biskra
Spécialité: Intelligence Artificielle

Vérification paramétrée à partir des spécifications formelles des systèmes d'information

Par: Sara Houhou

Soutenue le 22/12/2021 devant le jury composé de :

Ms. Barbara Re	Maître de conférence à Camerino Université, PROS	Rapportrice
M. Gwen Salaün	Professeur à Grenoble Alpes Université, LIG, CNRS	Rapporteur
Ms. Béatrice Bérard	Professeur à Sorbonne Université, LIP6, CNRS	Examinatrice
Ms. Lamia Hamza	Maître de conférence à Bejaia Université, LIMED	Examinatrice
M. Tibermacine Okba	Maître de conférence à Biskra Université, LESIA	Examinateur
M. Souheib Baarir	Maître de conférence à Paris Nanterre Université, LIP6, CNRS	Co-Directeur
M. Laïd Kahloul	Professeur à Biskra Université, LINFI	Directeur
M. Pascal Poizat	Professeur à Paris Nanterre Université, LIP6, CNRS	Directeur

A thesis submitted for the degree of Doctor of Philosophy at

Sorbonne Université

Mohammed Khider Université -Biskra-

École Doctorale EDITE de Paris (ED130)
Informatique, Télécommunication et Électronique

Laboratoire de l'Informatique Paris 6 (LIP6)
Laboratoire d'InFormatique Intelligente (LINFI)

Parameterised Verification from Formal Specifications of Information Systems

By **Houhou Sara**

Doctoral Thesis on Computer Science

Under the supervision of :

M. Laïd Kahloul	Professor at Biskra Université, LINFI	(Supervisor)
M. Pascal Poizat	Professor at Paris Nanterre Université, LIP6, CNRS	(Supervisor)
M. Souheib Baarir	Associate Professor at Paris Nanterre Université, LIP6, CNRS	(Co-Supervisor)

Presented on 22/12/2021 in front of a jury composed of:

Ms. Barbara Re	Associate Professor at University of Camerino, PROS	(Reviewer)
M. Gwen Salaün	Professor at Grenoble Alpes Université, LIG, CNRS	(Reviewer)
Ms. Béatrice Berard	Professor at Sorbonne Université, LIP6, CNRS	(Examiner)
Ms. Lamia Hamza	Associate Professor at Bejaia Université, LIMED	(Examiner)
M. Tibermacine Okba	Associate Professor at Biskra Université, LESIA	(Examiner)

To my father Abdesalem, who always supports me and believes in success in my life
To my mother Saida, who always pries for me to succeed in my life
To my brothers Ahmed, Akram, Aymen, and Amjed
To my beloved sister Manel
To my husband Tarek
To my nephews
To all my family
I dedicate this thesis to them

Acknowledgement

“*Saying thank you is more than good manners. It is good spirituality.*”

Alfred Painter

First and foremost, praises and thanks to my God, Allah, for everything.

I would like to express my sincere gratitude to my supervisors Prof. Laïd Kahloul and Prof. Pascal Poizat, and my co-supervisor Prof. Souheib Baarir, for accepting the supervision of this thesis. Thank you for all the guidance, patience, immense knowledge, and support you gave me at every stage in my PhD project. If I am in this stage today, it is thanks to you.

I express my profound gratitude to Prof. Philippe Quéinnec for his collaboration, his continuous support of my PhD study and related research, and all the help and support he gave me in each stage of this thesis. Thank you!

My sincere thanks also to the committee members: Prof. Gwen Salaün, Prof. Béatrice Berard, Prof. Barbara Re, Prof. Lamia Hamza, and Prof. Tibermacine Okba for accepting to evaluate this work.

I sincerely thank all the members of the LIP6 laboratory, especially the MoVe team, for the numerous helpful discussions during my thesis. I also thank my colleagues at the LIRMM laboratory, especially the MAREL team, for the innumerable helpful discussions during my stay in Montpellier. I also thank my colleagues at Paris-Nanterre University, especially Pr. Jean-François Pradat-Peyre for giving me the opportunity to teach at the SEGMI department. I sincerely thank all the members of the LINFI laboratory and my colleagues at the computer science department of Biskra university. Many thanks to my friends Samah Masmoudi, Dr Sara Hamouda, Dr Anas Shatnawi, Dr Rim Saddem, Dr Nassima Benammar, Dr Khadija Bousselmi, Dr Mostefa Bennaceur, and Dr Kamli Adel for all the help and the support they gave me in every stage of my research study.

Words cannot express how grateful I am to my father and mother for all their sacrifices for me. They have been with me throughout my life and whose love brought me where I am today. I owe a lot to both of you. I am very grateful to my brothers, sister, husband, and family members for all the support and help they provided throughout my life. Thank you very much.

Sara

Abstract

Process models are essential aids for learning, analysis, improvement and communication of a business process. BPMN is the leading standard in the context of business processes and workflow modelling languages. It provides a notation that is readily understandable by business users, ranging from the business analysts, who sketch the initial drafts of the processes, to the technical developers responsible for actually implementing them, and finally to the staff deploying and monitoring such processes. BPMN supports modelling using different types of diagrams, including the collaborative diagram. This diagram provides an efficient way to describe how several business entities, each with its internal process, can interact with one another to reach objectives.

Even if it is a widely accepted notation, the BPMN execution semantics is defined using natural language. This leaves room for interpretation and hampers the formal analysis of the process models. A great effort has been devoted to proposing formal semantics for BPMN and (fewer) providing dedicated verification tools. Still, some advanced features of BPMN, namely subprocesses, communication or time-related constructs, are often set aside. This becomes an issue as BPMN gains interest outside of its original scope, *e.g.*, for the Internet of Things (IoT), where communication and time play an essential role. The modelling of a process in BPMN and fully guaranteeing its behaviour may be complicated in the presence of such concepts. Thus, providing formal semantics taking into account the usual control flow-elements and subprocesses, inter-process communication, and time-related constructs is required. Further, the complexity of the provided process verification tools, their lack of a fully automatic support of the entire verification chain, the fact that they are not integrated into a process environment, and the impossibility for the average process modeller to express business properties, prevent their adoption.

This thesis faces these problems by providing a first-order Logic formal semantics for the BPMN collaboration diagrams that support the subprocesses, communication, and time constructs. It is parametric with reference to seven point-to-point communication models that exist when considering local and global message ordering. To perform verification, we have implemented this semantics in a tool suite called fbpmn. It allows performing the automatic verification of correctness properties of BPMN collaboration models and animating counterexamples if the properties are not satisfied. Our framework is built upon two different formal specification languages for the semantics implementation. A first implementation uses TLA$^+$ language, whose companion, the TLC checker model, supports explicit model checking. TLA$^+$'s expressiveness comes from being based on ZF (set theory), first-order logic, and configurable modules. A second implementation relies on the Alloy language, whose companion, the Alloy Analyser, provides support for bounded model checking. Alloy's expressiveness comes from being based on relational logic, first-order logic enhanced with the transitive closure operation, which renders the definition of structural properties extremely simple. Altogether, the fbpmn tool-suite, the TLC tool, and the Alloy Analyzer may check BPMN models for workflow specific properties. The fbpmn tool-suite is open source and freely available online.

Keywords— BPM, BPMN, Collaboration, Communication, Time, Formal Semantics, Verification, Framework, TLA$^+$, Alloy

Résumé

Les modéles de processus sont des outils essentiels pour l'apprentissage, l'analyse, l'amélioration et la communication autour d'un processus métier. BPMN est la norme standard pour la modélisation de processus métiers. Il fournit une notation compréhensible par les utilisateurs métier, allant des analystes métier qui désignent les ébauches initiales des processus aux développeurs techniques chargés de les mettre en oeuvre, et enfin au personnel qui déploie et surveille ces processus. BPMN supporte la modélisation à l'aide de différents types de diagrammes, parmi lesquels le diagramme de collaboration. Ce dernier fournit un moyen efficace de décrire comment plusieurs entités, chacune avec son propre processus interne, peuvent collaborer et interagir les unes avec les autres pour ôatteindre des objectifs.

Même s'il s'agit d'une notation largement admise, la sémantique d'exécution de BPMN est définit en langage naturel. Cela laisse place à l'interprétation et limite l'analyse formelle possible des modèles. Un gros effort a été consacré à proposer une sémantique formelle pour BPMN et, dans une moindre mesure, à fournir des outils de vérification dédiés. Cependant, certaines fonctionnalités avancées de BPMN, à savoir les sous-processus, la communication ou les constructions liées au temps, sont souvent laissées de côté. Cela constitue un problème car BPMN a suscité l'intérêt en dehors de son champ d'application-initial, par exemple pour l'Internet des objets (IoT) où la communication et le temps jouent un rôle important. La modélisation d'un diagramme BPMN, ainsi que la garantie complète de son comportement, peuvent s'avérer très difficiles à assurer en présence de tels concepts. Pour cela, il est nécessaire de fournir une sémantique formelle prenant en compte non seulement les constructions de processus habituelles, mais également celles liées aux sous-processus, à la communication inter-processus et au temps. D'autre part, la complexité des outils associés, leur absence de prise en charge entièrement automatique de l'ensemble de la chaîne de vérification, le fait qu'ils ne soient pas intégrés dans un environnement de processus et l'impossibilité pour le modélisateur moyen de processus d'exprimer des propriétés métier empêchent leur adoption.

Dans cette thèse, nous proposons des solutions à ces problèmes en fournissant une sémantique formelle en logique de premier ordre pour les diagrammes de collaboration de BPMN qui prend en charge les constructions liées aux sous-processus, à la communication et au temps. Cette sémantique est paramétrique par rapport sept modéles de communication point à point qui existent lorsque l'on considère l'ordre local et global des messages. Nous avons implémenté cette sémantique dans une suite d'outils appelée fbpmn. Elle permet d'effectuer la vérification automatique des propriétés de correction pour les modèles de collaboration BPMN et d'animer les modèles des contre exemples lorsque les propriétés ne sont pas satisfaites. Notre cadre logiciel est basé sur deux languages de spécification formelle différents. Une première implémentation de la sémantique utilise le langage TLA$^+$. Il est accompagné de plusieurs outils, dont le model checker TLC qui prend en charge la vérification de modèle explicite. L'expressivité de TLA$^+$ vient du fait qu'il est basé sur ZF (théorie des ensembles), la logique du premier ordre et des modules paramétrables. La deuxième implémentation de la sémantique est basées sur le langage Alloy. Il est accompagné d' Alloy Analyser qui prend en charge la vérification de modèle bornée. L'expressivité d'Alloy vient du fait qu'il est basé sur une logique relationnelle, une logique de premier ordre renforcée par l'opération de fermeture transitive, ce qui rend la définition des propriétés structurelles extrêmement simple. Tous ces outils, l'outil fbpmn, l'outil TLC et Alloy Analyser peuvent être utilisés pour effectuer la vérification des propriétés spécifiques aux workflows de modéles BPMN. La suite d'outils fbpmn est open source et disponible gratuitement en ligne.

Mot Clés— BPM, BPMN, Collaboration, Communication, Temps, Sémantique Formelle, Vérification, Outil, TLA$^+$, Alloy

Contents

Abbreviations

BP	Business Process
BPM	Business Process Management
BPMN	Business Process Modeling Notation
XML	Extensible Markup Language
FOL	First Order Logic
TLA	Temporal Logic of Actions
LTL	Linear Temporal Logic
CTL	Computation Tree Logic
FBPMN	Formal Business Process Modelling Notation

List of Figures

List of Tables

Chapter

1 Introduction

Chapter content

1.1 Thesis Context

Business process management (BPM) focuses on the modelling and management of business processes by using suitable techniques that allow organisations to be more efficient and flexible in achieving their goals. Business process models are the key instruments of BPM. They explicitly represent the BPs in terms of their activities and the execution constraints between them. A Business Process (BP) may be automated in whole or in part by a software system. According to the temporal and logical dependencies set in an underlying process model, an automated business process (also known as workflow) passes information from one participant to another for action. The system works with automated business processes called Process-Aware Information System.

Process-Aware Information Systems (PAISs) [1] are information systems that link Information Technology to Business Processes. Dumas *et al.* in [2] define a PAIS as *"a software system that manages and executes operational processes involving people, applications, and/or information sources on the basis of process models"*. The term process model, by definition, refers to a business process representation using some graphical notation. In fact, in a PAIS (cf. Figure 1.1), process modelling is the initial step of the business process lifecycle, called *design*. In this step, high-level business requirements are designed as a process or a workflow. The models can be designed at different levels of abstraction, typically through business process modelling notations and their supporting tools. A range of graphical notations have been proposed for business process modelling, such as Business Process Model and Notation (BPMN) [3], Event-driven Process Chain (EPC) [4], Yet Another Workflow Language (YAWL) [5], Unified Modelling Language (UML) [6], etc. The second phase concerns the *implementation* of these business process models. In this step, the process models are automated into executable processes by refining the models into operational process specifications and joining the different process activities to concrete applications and organisational entities. This can achieve by using PAIS systems such as Workflow Management System (WfMS) or Business Process Management Systems (BPMSs). The third step is the *execution* of these process models. Here, the process model can be deployed to a process engine to be executed. The final step is the process *diagnosis*. Here, the operational processes are analysed to identify possible problems and find aspects that can be improved. The feedback of this phase may be used to redesign the processes, and the cycle continues [7, 8].

A BPMS is a system that supports the design, analysis, execution, and monitoring of business processes based on explicit process models [8]. BPMSs originate from WfMS (*i.e.*, an older type of PAIS), which are focused on modelling and execution steps and did not very well support the other phases of the BPM's lifecycle (cf. Chapter 2.2). In contrast, BPMS fully support the entire BPM lifecycle. Furthermore, BPMSs are the automated implementation of the business process models. Thus, the incorrectness of the process models may directly impact the core of the business operations. For that, the ability to detect errors in the early design phase and analyse process models is likely to become a desirable feature for tools supporting process modelling. Wherefore, without tools offering modelling and automatic

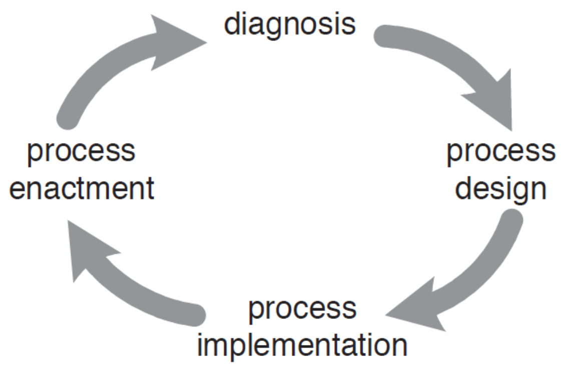


Figure 1.1: *The PAIS Lifecycle. (from source figure [9])*

verification of process models, a modeller expert can not guarantee the efficiency and reliability of the process models.

To tackle this problem, the development of business process systems requires a shift to formal methods to increase the designed solution's reliability and avoid unexpected effects after their implementation. The use of formal methods allows the designers to verify their process design's correctness and detect potential errors. If it is the case, the processes can be improved before implementation. One of the techniques used to check the correctness of such systems is Model Checking (MC) [10]. Formally describing a business process requires tackling several challenges. Among them, the business process models need to be viewed from several distinct perspectives: *control flow, interaction, data, resource*, and *time*. The formal model must take into account these perspectives. Thus, this thesis proposes a formal framework for supporting the control flow, the interaction, and the time perspectives of business processes models in BPMs based on BPMN as a graphical modelling language. The context of this thesis supports the two fundamental phases of the BPM's lifecycle, which are the process *Design* and *Analysis.*

The rest of the chapter introduces the motivations behind the research presented in this thesis, the research questions it aims to answer, the research contributions of the thesis, and finally, the thesis structure.

1.2 Motivation and Problem Statement

BPMN became the most prominent notation for representing business processes, thanks to its wide usage in academic and industrial contexts [11]. It is an ISO standard notation with good tooling support (cf. Chapter 2.3). It allows business process designers to model both intra-organisational processes (single processes) and inter-organisational collaborations where communication coordinates processes in different organisations. This modelling can be achieved through the use of *process* and *collaboration* diagrams. A business process diagram in BPMN defines as a sequence of activities, events, and gateways (also called *flow element*), connected by sequence flows (also called *control flow)*, that denotes their ordering relations. The BPMN standard defines a theoretical concept, called a *token*, traversed the process structure from its start element to its end element, used as an aid for defining the process behaviour when it is performed. The behaviour of a process element can be defined by describing how it interacts with the tokens. Depending on the semantics of that traversed flow element, the number of tokens in a process can vary, as they are continuously generated and consumed. As an example, a start event generates a token, while an end event consumes one. A collaboration diagram in BPMN, defines the interaction amongst processes, using message flow element constructs (message activities and message events) connected by message flows. In collaboration, the behaviour of a process element triggers message flow to produce messages and its internal behaviour.

The fact that BPMN models are implementable and integrate constructs for transmitting messages between processes makes their analysis and ensures their correctness before using them in a real context a challenge. However, the BPMN standard is related to using the semi-formal definition to define its execution semantics. This semantics is described in a semi-formal language and meta-model, and it is dispersed through the specification. This leaves room for interpretation and hampers the formal analyses that would be desirable to find defaults at design time rather than when running the processes and collaborations over business process engines. The current state-of-the-art has already proposed a formalisation of the BPMN execution semantics. Indeed, attention has been raised to provide formal semantics for collaboration diagrams to capture the features of message exchanges and data (cf. Chapter 3). Despite all these efforts, these proposals leave apart essential features to consider when dealing with collaboration diagrams which are the communication models between the nodes of the system and its configuration in different communication modes. Meanwhile, BPMN is gaining interest as a modelling language in new areas, such as the Internet of Things (IoT) [12, 13] where the communication and time perspectives present its fundamental.

The time perspective is a critical dimension to consider as it is closely related to customer satisfaction and cost reduction [14]. Time plays a role in negotiating frequent delays in outsourcing, ensuring the completion of activities on time and the availability of the final product on time. The business process field is influenced by a wide range of temporal constraints, which rise from constitutional, regulatory, and managerial rules. Thus, BPMN defines a set of time-related elements (timer flow element). Each of these timer elements defines time information according to its kind of TimerEventDefinitions [3]. However, this time aspect is poorly addressed in the current standard initiatives. No explicit formal semantics for BPMN time-related features is given in the standard. Further, to associate time information to such elements. BPMN relies on the ISO-8601 standard time definitions. The ISO-8601 standard is too reach and its description of time information is quite complex. In general, it defines three kinds of TimerEventDefinitions (*timeDate*, *timeDuration*, and *timeCycle*) with a different format of configurations. This makes it more difficult to perform a formal analysis of process models. Indeed, the absence of a precise semantic for BPMN timer elements reduces the models' comprehension. For that, most of the previous attempts of formalising the BPMN time constructs execution semantics is based on the introduction of additional modelling features to specify the temporal aspects (cf. Chapter 3) and leave apart the features related to time-related events in the broader sense. In addition, the BPMN model defines different temporal constructs. This may result in a model with complex scenarios in which it is impossible to simultaneously satisfy all temporal perspectives of the process model. To analyse such cases and detect inconsistencies, formal semantics of the temporal concepts used for specifying the time perspective of a BPMN process model is required. Further, a lot of effort has been undertaken to identify the most common time constraints in the business perspective, which have been called Process Time Patterns [15]. The time patterns provide a universal and comprehensive set of notions for describing the temporal aspects of business processes and eliciting fundamental requirements. In the literature, works focus on describing the time patterns semantics specification using natural language and independently of any specific process modelling language (cf. Chapter 3). However, less attention has been paid to assessing the suitability of BPMN to express these time patterns graphically, and no formal semantics for these patterns specified in BPMN was provided.

For that, this research work aims to answer the following questions that are strictly related.

Q1: Does the correctness of BPMN collaboration diagrams depend on the used communication models?

Q2: How to precisely describe the formal semantics of BPMN collaboration diagrams taking into account different communication models?

Q3: How to formalise the execution semantics of the BPMN time constructs, including their relation to the ISO-8601 standard format?

Q4: What are the time process patterns supported by the BPMN standard, and does our semantics support all of them?

Q5: How to verify such formal models?, which are the properties of interest?, and can a formal semantics of the BPMN collaborations drive the development of software tools based on BPMN collaboration diagrams?

Roughly speaking, regarding all the previous requirements, a formal framework considering precise semantics for the communication and time, enabling an exhaustive and automatic verification of the BPMN models is crucial before performing business processes.

1.3 Research Contribution

In light of the shortcomings mentioned above, the core objective of this thesis is to provide a systematic methodological approach to improve the modelling of the BPMN process and collaboration diagrams. We intend to introduce a formal framework that allows novice and experienced BPMN designers to understand their models and properties better. The approach contributes a generic formalisation that supports the control flow elements, communication, and time perspectives of BPMN models. Furthermore, it provides a foundation technique for modelling and verifying the consistency of the execution semantics of the BP models, taking into account distinctive characteristics introduced by different communication modes and modelling with different time-aware process constructs. The main research contributions can be summarised as follows:

- A precise formal semantics for a subset of BPMN elements, including *control flow elements, subprocess, message exchanges, communication modes* for modelling process and collaboration diagrams, is provided. This semantics is compliant with the operational execution semantics described in the standard [3]. The formal semantics is defined on BPMN elements in terms of First-Order Logic (FOL), rather than encoding into other formalisms. The FOL semantics provides a universal and comprehensive set of notions for describing business processes' execution behaviour and eliciting fundamental requirements. Further, to foster the use of the BPMN collaboration models in a wide range of application scenarios (*e.g.*, IoT systems), a modular structure for incorporating seven generic communication models relating to message-passing behaviours between and within processes and their formal semantics definition is provided. Moreover, the modular structure includes ad-hoc communication models (a specific model built by assembling micro communication models that offer different sending and receiving messages constraints). The formal definition of these models will foster the integration of the communication models into BP systems, significantly widen the application scope of BPM.

- A precise formal semantics for BPMN time-related constructs is provided. This semantics is based on an in-depth analysis of the time definition types (date-times, durations, and cycles) regarding the ISO-8601 formats [16] as specified in the BPMN standard [3]. The formal semantics allows avoiding ambiguities regarding the use of temporal concepts and helping to detect or predicate possible inconsistencies or critical situations that may occur during run time (*e.g.*, excessive delays). Moreover, a well-founded set of time patterns representing temporal concepts in BPMN is collected from the literature. These time patterns provide a universal and comprehensive set of notions for describing the temporal aspects of BPMN business processes and eliciting fundamental requirements. In particular, we discuss how the provided semantics support each time pattern's related BPMN process model.

- A generic framework that supports the formal semantics, including the communication models, is provided. This framework performs the verification of correctness properties for business process and collaboration models automatically. The implementation of the semantics is given in TLA$^+$ [17] and Alloy [18] as a set of theories. This corresponds to a pure syntactic transcription of the FOL into the corresponding TLA$^+$ $\setminus$ Alloy fragments. Moreover, it supports the verification of a novel variant of BP properties introduced for the BPMN collaboration diagrams. Moreover, intending to give more flexibility and freedom for the designer, the framework supports an arbitrary topology of business process models. It does not impose any syntactical restriction on the usage of the modelling notation, such as well-structureless (*e.g.*, the forbidden use of mixed gateways format). Farther, the framework considers advanced aspects: (i) including the support of the ad-hoc communication models, (ii) using the animation to illustrate situations where a BPMN schema is found to violate a given property, and (iii) supporting two different notions of time: an abstracted version encoded in TLA$^+$ and an explicit one encoded in Alloy.

Altogether, the results presented in this thesis aim at (1) provide a formalisation of a subset of BPMN execution semantics that supports control flow, time, and interaction and that is parametric with reference to the properties of the communication between participants, (2) support this formalisation with tools that automatically perform the verification of correctness properties for the BPMN process and collaboration models.

1.4 Thesis Structure

This doctoral thesis is divided into four parts and seven chapters. Bibliography and appendices complete these parts.

Part I - State of the Art.

- **Chapter 2 - Background** presents the necessary background information by introducing the process modelling languages as well as the formal descriptions and notations used throughout the thesis. In practice, BPMN, FOL, TLA$^+$ and Alloy languages are presented.
- **Chapter 3 - Literature Review** positions our work by reviewing the existing literature on the formalisation of BPMN business process models addressing the support and the verification of business processes dealing with the communication and the temporal aspects.

Part II - BPMN 2.0 Semantics Formalisation. This part is made up of two chapters that gathers our semantics.

- **Chapter 4 - BPMN and Communication** provides a formal semantics for the BPMN collaboration diagram. First, it defines seven possible communication models relating to message-passing behaviours between and within processes and provides their formalisation using FOL. Second, it presents the FOL semantics of each BPMN element, taking into account these communication models. Finally, it shows how you can interchange them when studying a given BPMN schema.
- **Chapter 5 - BPMN and Time** provides the formalisation of BPMN time construct in the presence of an explicit version of time considering the ISO specification for date and time, time intervals, and recurring intervals. Then, it presents its support for eight-time patterns in BPMN.

Part III - From Formal Semantics to Tool Support. This part focuses on the implementation of the proposed formalisations.

- **Chapter 6 - fbpmn: Formal BPMN Framework** presents a framework for verifying a large class of BPMN schemas against many classical properties. It shows the implementation of the formalisations into two formal languages TLA$^+$ and Alloy, the verification process, and the evaluation of the tool under a set of experiments.

Part IV - Conclusion and Future Work. This part concludes the work.

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

1.5 List of Publications

- **Sara Houhou**, Souheib Baarir, Pascal Poizat, Philippe Quéinnec, and Laid Kahloul. A First-Order Logic Verification Framework for Communication-Parametric and Time-Aware BPMN Collaborations. Information Systems, 2021. (Core: A)
- **Sara Houhou**, Souheib Baarir, Pascal Poizat, and Philippe Quéinnec. A Direct Formal Semantics for BPMN Time-Related Constructs. In: 16th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE) 2021. (Core: B, *Best Student Paper Award Nominate*)
- **Sara Houhou**, Souheib Baarir, Pascal Poizat, and Philippe Quéinnec. A First-Order Logic Semantics for Communication-Parametric BPMN Collaborations. In : 17th International Conference on Business Process Management (BPM), p. 52-68. Springer, 2019. (Core: A, *Best Paper Award*)

Part I

State of the Art

Chapter 2
Background

“The only way of discovering the limits of the possible is to venture a little way past them into the impossible.”
Clarke’s Second Law

Chapter content

2.1 Introduction

This chapter aims to provide an overall understanding of the background notations relevant to the dissertation content. Firstly, it presents the concepts and technologies relevant to business process management using a business process life-cycle. Secondly, it introduces the standard modelling notation BPMN 2.0, and it gives two BPMN collaboration diagrams scenarios to be exploited through the rest of the thesis. Then, it presents the existing verification methods. Finally, it provides an overview of the First-Order Logic specification, the TLA$^+$ and Alloy formal languages. These later are increasingly adopted and have been successfully applied to complex systems (*e.g.*, the works in [19] and [20]).

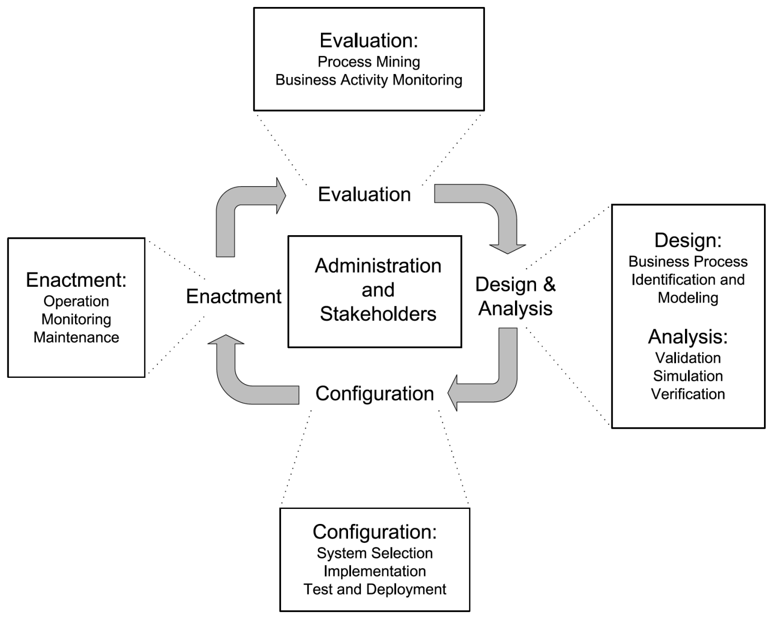


Figure 2.1: *BPM Lifecycle According to Weske. (from source figure [8])*

2.2 Business Process Management (BPM)

Business Process Management (BPM) is a discipline for improving and optimising business processes, centred on the processes integration and management to achieve business goals. BPM was founded to develop approaches to the operationalisation of business processes based on software technologies. BPM is described by Dumas *et al.* as *"A body of methods, techniques and tools to identify, discover, analyse, redesign, execute, and monitor business processes to optimise their performance.[1]"*. In other terms, M.Weske define the BPM by *"A Business process management includes concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of business processes.[21]"*

BPM is about improving and managing a set of events, activities, and decisions that ultimately add value to an organisation and its clients. These successions of events, activities, and decisions are called *processes* [1]. The key instrument of BPM is the *Business Process (BP)* notion. M.Weske defines BP as *"A business process consists of a set of activities that are performed in coordination in an organisational and technical environment. These activities jointly realise a business goal. Each business process is enacted by a single organisation, but it may interact with business processes performed by other organisations. [21]"* With the same insight, M.Dumas.*et al.*, define a business process as *"a collection of inter-related events, activities, and decision points that involve a number of actors and objects, which collectively lead to an outcome that is of value to at least one customer. [1]"*. In general, BP is a particular type of process, defined as a set of tasks that need to be executed in a specific order to realise one or more business goals. The business processes describing an internal behaviour limited to one organisation are called, *intra-organisational processes*. Whilst those interacting with business processes, organisations, for example, providing information to them based on their requests, are called *inter-organisational processes*.

A set of phases characterises BPM, noted by a *BPM lifecycle*, that occurs cyclically to adapt and improve the model. According to M.Weske [8], the lifecycle of BPM is composed of four main phases, as shown in Figure 2.1.

- **Design and Analysis.** Firstly, the business process is identified, and a model representing the process is manually designed and/or automatically elicited. The obtained process model is then validated by assessing its formal correctness, simulating all the possible executions, estimating costs, execution time and resources allocation. All these analyses are carried out based uniquely on the model without actually executing the process. In addition, the model is manually inspected to verify that all the relevant aspects characterising the process are captured in the model.
- **Configuration.** Once the process model is defined and validated, it is implemented. This can be done in different ways. The process model can be refined to become executable by a BPMS (a software component responsible for coordinating the execution of the process according to the

model). Alternatively, ad-hoc software that behaves according to the process model can be developed. Finally, the process model can derive a set of policies or guidelines that human operators should follow. It is worth noting that, depending on how the process is implemented, the execution of the process can be completely, partially or not be carried out automatically and autonomously. Generally speaking, whenever human operators are involved in executing activities, the process is not completely automated.

- **Enactment.** Once the process is implemented, it can be executed. While the process is running, it is monitored to identify when actual executions take place (*i.e.*, an instance of the process is created) and terminate when activities composing the process are started and concluded, and if exceptions occur during the execution of the process or a specific activity. This information is usually collected and stored in log files. If the process is (partially) automated, the BPMS (or the ad-hoc software) makes sure that activities are executed at the right time.
- **Evaluation.** Once the process is enacted, the real efficiency and effectiveness of the process are assessed. For instance, the average time spent running each activity can be determined, as well as the idle time. Possible bottlenecks in the execution and allocation of activities can be identified. Additionally, it can be determined to which extent the process model reflects the actual execution of the process. Such information can then be used as an input for the Design and Analysis phase, thus completing the cycle.

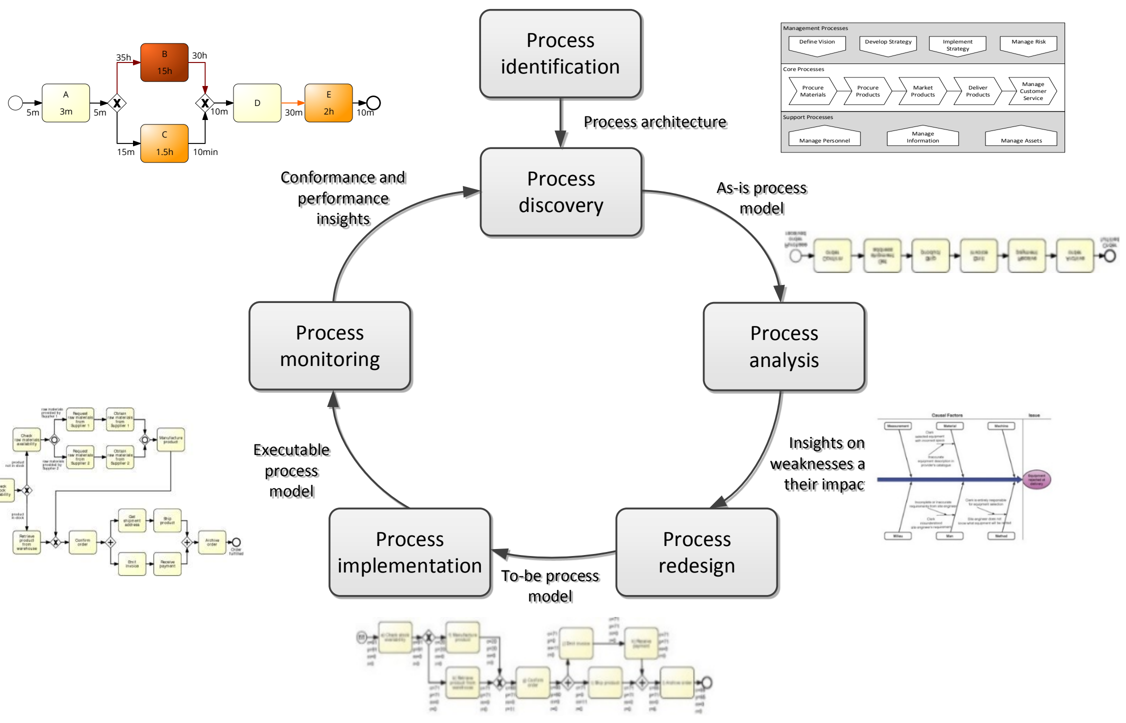


Figure 2.2: *BPM Lifecycle. (from source figure [1])*

In [1], Dumas *et al.* describe the lifecycle of BPM, as shown in Figure 2.2. However, they propose a slightly different classification of the phases characterising such a lifecycle. The figure distinguishes six major management activities for business processes.

- ***Process identification.*** In this phase, the modeller describes the overall process organisation as a process architecture.
- ***Process discovery.*** In this phase, the modeller use discovery techniques (*e.g.*, observation, document analysis, automated process discovery, etc.), modelling notations (*e.g.*, BPMN, DMN, etc.),

and tools that include (syntax, style checking, model repository management, and dictionary) for discovering, decorticating, and capturing the process architecture as a business model. This phase comes up with an *as-is process model* that describes the process (*i.e.*, how it currently is and who are involved in the process on a regular basis).

- ***Process analysis.*** In this phase, the modeller uses the *as-is process model* and the associated documentation for identifying, quantifying weaknesses (*i.e.*, issues) and the impact on the performance of the process by using them.

- ***Process redesign.*** In this phase, the modeller modifies the *as-is process model* according to the resulting insights and weaknesses that are identified in the analysis phase and comes up with a *to-be process model.* This phase is about making a trade-off between different performance dimensions, for which we find: time, cost, quality, and flexibility. The art of the redesign phase is to improve the model with one of these dimensions without affecting the other dimensions.

- ***Process implementation.*** In this phase the *to-be process model* is putting into execution. To establish this step, two actions are required: (i) Providing an information systems infrastructure for running the process. (ii) Educating and training people to perform the to-be process model or putting in place a system for continuously managing the business process's performance.

- ***Process monitoring.*** This phase concerns the process performance and conformance observation to insight any unexpected impact of the changes made in the process. In case of the non-achieved objectives stated in the redesign phase, the modeller can proceed with another cycle of process discovery, analysis, redesign, and implementation.

As the main focus of this thesis is on business process design and analysis, the following sections define a set of notions used in these phases.

2.3 Business Process Modelling Language (BPMN)

A range of graphical process modelling languages have been proposed to represent business process models such as BPMN [3], Petri nets (PN) [22], Event-driven Process Chains (EPCs) [4], YAWL [5], UML activity diagrams [6], etc. Without limiting the generality of our work, we select and use BPMN as input notation.

BPMN is the leading standard in the context of business processes and workflow modelling languages. The primary goal of BPMN is to provide a notation that is readily understandable by business users.

BPMN was released in 2004 by the Business Process Management Initiative (BPMI) [3] as a graphical notation (partially inspired by UML Activity Diagrams) to represent the graphical layout of business processes. BPMN is highly adopted by business analysts and has acquired a clear relevance among the notations used to model business processes in academia and industry. The ever-increasing number of adoptions from companies and the growing interest in this notation caused the adoption of BPMN as an OMG standard in 2006. In 2013, BPMN 2.0 was named as an ISO/IEC standard for modelling business processes. BPMN has different versions from (1.0, 1.1, 1.2, 2.0). BPMN 1.X, ($X \in 0, 1, 2$) versions did not have a clearly defined semantics nor a native serialisation format. This is why we focus on the newest major version of BPMN, namely 2.0. The BPMN 2.0 specification extends the scope and the capabilities of the BPMN 1. X in several areas (execution semantics for all BPMN elements, a wide collection of constructs, modelling interactions, etc.).

BPMN provides a graphical notation representing a business process as a Business Process Diagram (BPD). It defines four main kinds of diagrams: *process, collaboration, choreography, and conversation diagrams.* This thesis focuses on the two first. A process diagram is used to model the activities of a single organisation. Collaboration diagrams can be defined with different processes (for various organisations), exchanging messages and cooperating to reach a shared objective. This thesis abstracts away from data (data objects, data stores and message payloads). Therefore, message (resp. event instance) and message type (resp. event type) are used interchangeably in the sequel. For a complete and detailed description of each BPMN diagram, please refer to the official BPMN specification [3].

The following sections describe a subset of BPMN elements notation and then introduce how to employ BPMN notations as a running example for the rest of the document.

Figure 2.3: *BPMN Pool.*

2.3.1 BPMN Notation

BPMN is the well-known diagrammatic notation for supporting the specification of business processes. The notational elements of BPMN are classified into four groups: *Flow objects*, *Connecting objects*, *Artefacts* and *Swimlanes*. The flow objects and connecting objects are the basic elements for constructing business processes. The extra information and the organisation perspective of a business process diagram are expressed using artefacts and swimlanes. As the artefacts and the swimlanes are unrelated to the process flows and do not have a semantics-based execution, we describe only the elements considered in the thesis.

1. **Pools** group a set of activities with some common characteristic, *e.g.*, a specific role or a process participant capturing the resource perspective. They represent a participant or organisation as a process diagram that may be involved in collaboration with other processes to represent a collaboration diagram. Graphically, a pool is represented by a rectangle with a specified name referring to an organisation (see Figure 2.3).

2. **Flow Objects** are the basic graphical elements that allow defining the behaviour of the BPMN model. The flow objects fall into three categories: *Events*, *Activities*, and *Gateways*.

   - Events represent facts that occur instantaneously during process execution and affect the sequencing or timing of process activities. They are drawn as circles, which may contain markers to diversify the kind of the event trigger (see Figure 2.4). A process requires a starting point and a termination point. In BPMN, these are identified by the *Start events* and the *End events*. A *Start event* is the start point of the process that initiates a new process instance. An *End event* ends the flow of activities; that will typically (under certain conditions) terminate or complete a process instance.

     In addition, this set of elements is called by the event because it may (optionally) trigger an event, *e.g.*, send a message to another process or environment. Different events are triggered during the execution of a process instance, *e.g.*, the arrival of a command order that neither starts a new process instance nor terminates one. This event affects the process flow in the sense that it must occur for the process to go on, called by *Intermediate events*. BPMN defines a set of trigger types (*e.g.*, Message, Signal, Timer, etc.) It associates to each event an *eventDefinition* to determine its trigger type. The *eventDefinition* may have one or more trigger types. If its *eventDefinition* is empty, then the event node type is *None*. If the *eventDefinition* has more than one trigger, the event node type is *Multiple* (this type of event is out of the scope of this thesis). In this thesis, we focus on the following event types:

     – *None Start Event* represents the initiating point of a process instance without any condition.
     – *Message Start Event* initiates a process instance when a message is received from an external participant.
     – *Timer Start Event* initiates a process instance when a fixed time is elapsed.
     – *Message Intermediate Throw Event* sends a message to an external participant.
     – *Message Intermediate Catch Event* receives a message from an external participant.
     – *Timer Intermediate Catch Event* represents delays expected within the process.
     – *Boundary Event* is an intermediate event that is attached to the boundary of an activity. There are two kinds of boundary events: *interrupting intermediate events* that interrupt the activity they are attached to. Whereas *non-interrupting intermediate events* initiate a new process path (*i.e.*, launch activities in parallel). The behaviour of these events is based on some condition (*e.g.*, a message reception or the deadline of a timeout).

  - *None End Event* terminates the process instance.
  - *Message End Event* sends a message to an external participant.
  - *Terminate End Event* indicates that all activities in the process should be immediately ended.

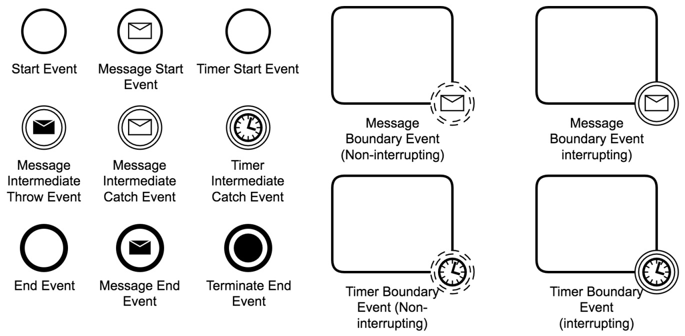

Start Event
Message Start Event
Timer Start Event
Message Boundary Event (Non-interrupting)
Message Boundary Event interrupting)
Message Intermediate Throw Event
Message Intermediate Catch Event
Timer Intermediate Catch Event
End Event
Message End Event
Terminate End Event
Timer Boundary Event (Non-interrupting)
Timer Boundary Event (interrupting)

Figure 2.4: *Considered BPMN Events.*

Furthermore, the term *event* in the BPMN standard has many interpretations. It denotes an event node (catching or throwing event) or an event in a closer sense (*i.e.*, a particular occurrence of something at a specific time). In this thesis, the term *event* denotes an event node under the primary use of the word in the BPMN standard. The term trigger indicates the occurrence of something in case of catching. In contrast, it means a result in a throwing case.

- Activities identify work that is realised in a process. There are two kinds of activities: an elementary atomic unit of work that can not be broken down to a more acceptable level of abstraction, called *Task*, and compound activities whose internal details are modelled using other elements, called *subprocess*. Tasks are drawn as rectangles with rounded corners having a label that specify their name. Subprocess can either be represented as collapsed, *i.e.*, as a task decorated by an +' sign, or can be expanded to show internal details (see Figure 2.5). Activities allow applying different actions under various circumstances. For that, BPMN provides different task types. In this work, we are interested in the following types:
  - *Abstract Task* represents the performing of an action.
  - *Send Task* represents the performing of an action involving the sending of a message.
  - *Receive Task* represents the performing of an action involving the receiving of a message.

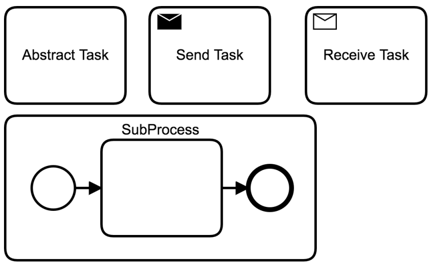


Figure 2.5: *Considered BPMN Activities.*

- Gateways are used to control the divergence and convergence of the sequence flow, particularly the activity execution order. Graphically, a gateway is drawn as a diamond with an internal marker that differentiates their routing behaviour (See Figure 2.6). There are five main types of gateways in BPMN, and we are taking into account four of them.
  - *Exclusive Gateway*, denote by ◇ symbol is a routing point in the process flow where it is used to choose one of the sets of mutually exclusive alternative incoming or outgoing branches. The choice of the outgoing branches is based on the evaluation of a data-based condition.

- *Parallel Gateway*, denote by symbol, is a routing point in the process flow where it synchronises concurrent flows for all its incoming branches and creates concurrent flows for all its outgoing branches.
- *Inclusive Gateway*, denote by symbol, is a routing point in the process flow where it has two specific behaviours. It synchronises two or more concurrent incoming edges and creates concurrent flows for all or some of its outgoing edges according to their condition evaluation.
- *Event-based Gateway*, denote by symbol, is a routing point in the process flow where event occurrence (time elapsing or message receiving) determines which path to follow, and all the others are discarded.

The four gateway types (exclusive, parallel, and inclusive) can be merging, splitting, or mixed (both merging and splitting).

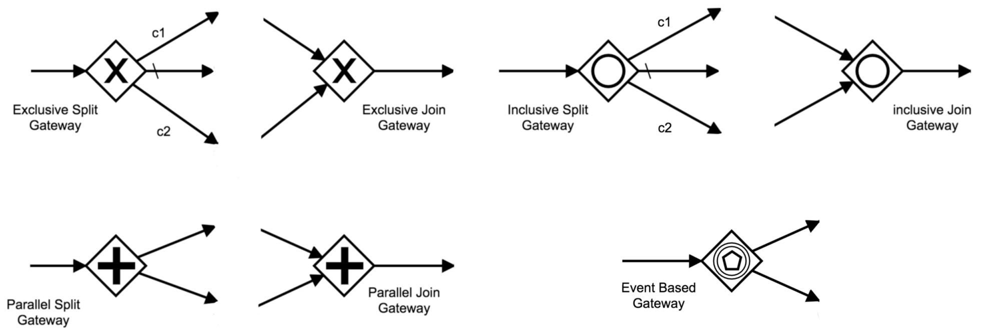


Figure 2.6: *Considered BPMN Gateways.*

3. **Connecting Objects** are used to connect flow objects to each other or to the artefacts. BPMN defines three main kinds: *Sequence flow*, *Message flow*, or an *Association flow* (Figure 2.7).

   - Sequence flow shows the execution order of flow elements. This category can be decomposed into normal sequence flows, conditional sequence flows (expressing the condition for some branch to be activated), and default sequence flows (the default branch to activate if all others (conditional ones) cannot be).
   - Message flow determines the message following between pools.
   - Association flow allows to associate artefacts to a flow or to connect them to an activity. The association flow is out of the scope of this thesis.

2.3.2 Running Example

As motivated in Chapter 1, both the specification and operational semantics support of the communication and the temporal constraints constitute fundamental challenges for the BPMN models. To help the reader get familiar with the BPMN modelling activity and discuss some of the challenges emerging in these contexts, we consider two examples consisting of two collaboration diagrams. This section presents for each of them the business scenario and the corresponding collaboration diagrams.

Travel Agency Collaboration Diagram. This example introduces a scenario concerning booking travel. We use this scenario to motivate our approach through Chapters 4 and 6. The presented example is derived from the one presented in [23]. The scenario involves two participants that act in a collaboration diagram. These are:

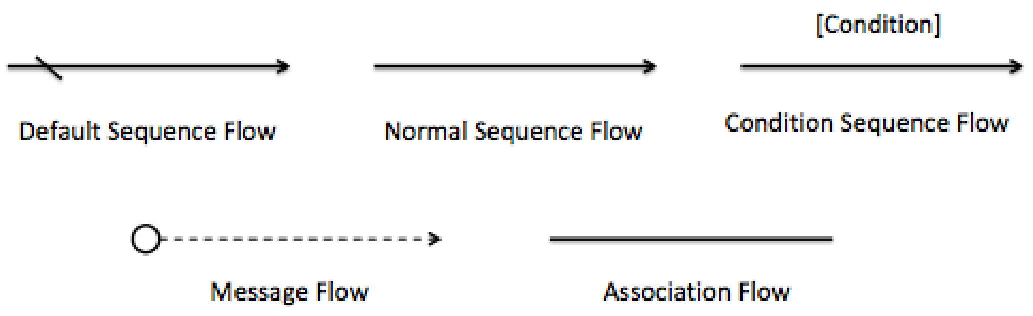


Figure 2.7: *BPMN Connecting objects.*

- **Customer** is a person who performs the booking travel request.
- **Travel agency** is an application that performs the confirmation of the travel reservation.

Figure 2.8 shows the collaboration diagram of the booking travel procedure. The collaboration starts when the *Customer* sends a request for an offer to the *Travel agency*.

- The *Travel Agency* sends offers to the client (loop with an exclusive gateway) in a first subprocess. Then, it starts a second subprocess to exchange information (booking, payment, confirmation, ticket) with the *Customer*.
- The *Customer* may reject some of the offers, and at some point, he/she may accept one (loop using two exclusive gateways).
- If so, he/she stops accepting offers, informs the *Travel Agency*, and sends the booking and the payment information to the *Travel Agency* and gets the corresponding tickets and confirmation. The *Travel Agency* and the *Customer* rely on interrupting features to deal with the fact that the *Customer* stops accepting offers as soon as he/she has agreed on one.
- The *Travel Agency* has only a fixed number of offers to send. Therefore, if the *Customer* has not agreed on an offer before the end of this set, the sending of offers by the *Travel Agency* will stop. In addition, the *Travel Agency* will send an interrupting message to the client (message boundary event on the customer offer reception task), which will interrupt the exchange subprocess (message boundary event on the second agency subprocess).
- If the *Customer* has accepted an offer before the end of the offers, he/she will start to send information to the *Travel Agency*. The *Travel Agency* will interrupt the sending of the set of offers and continue on the subprocess of the booking process

As one can see, the *transaction between communication, interrupting features* makes the overall behaviour of the collaboration challenging to grasp. Will the collaboration always reach one of its ends? Either that the client and the agency have agreed on an offer (them ending in Transaction Completed and Offer Completed, respectively) or that they have not waited for some event to happen (ending in Transaction Aborted and Offer Aborted, respectively). Will the collaboration reach one of its ends but with pending messages that have been sent but neither received nor treated? Or, worse, will the collaboration deadlock at some point depending on the choices made by the *Customer* and the *Travel Agency*, and the passage of time?

Paper Reviewing Process Collaboration Diagram. This example introduces a scenario concerning the management of a paper reviewing process of a scientific paper sending for a special issue in a journal. We use this scenario to motivate our approach through Chapter 5. The presented example is inspired by the one described in [2, Figure 4.79, Section 4.7]. The scenario involves three participants that act in a collaboration diagram. They are :

- **Journal PC Chair,** who performs the assignment of the submitted paper and managing the final decision. He is a person who edits the journal.
- **Author,** who performs the writing of the research paper and its submission. He is a person who specialises in the journal topic domain.

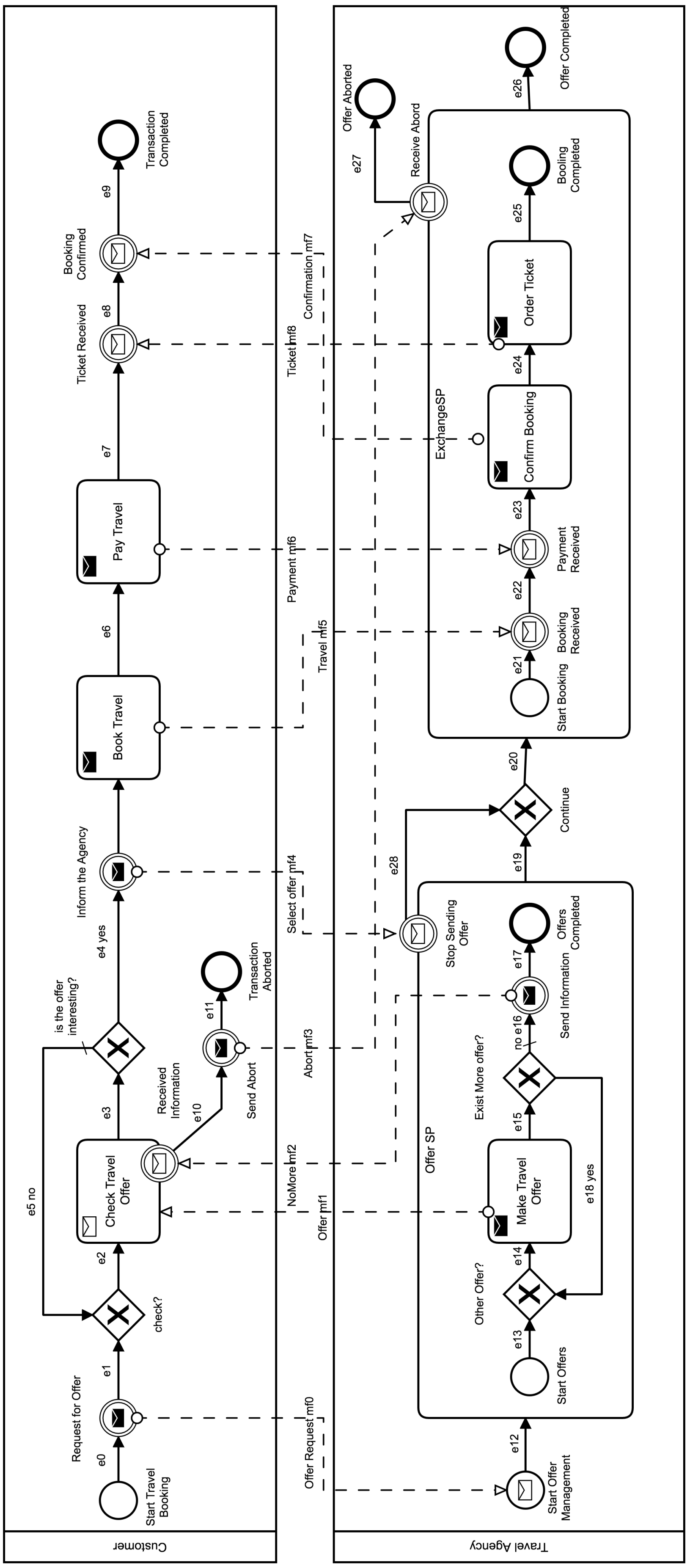


Figure 2.8: *Travel Agency Case Study. (extended from an example in [23])*

- **Reviewer,** who performs the reviewing process. He is a person with knowledge in the journal special issue topic.

Figure 2.9 presents the collaboration diagram of the reviewing procedure. For simplification, we consider only one author and only one reviewer. The collaboration starting when the submitted date is reached, and the author sends a paper to the *Journal PC Chair* through the submit paper send task.

- The *Author* will wait until the arrival of the notification response. If he/she doesn't receive a notification by 120 days from the sending date, the author withdraws the paper.
- The *Journal PC Chair* starts when the specified date and time, (*2021-01-17 T 00:00:00*), of the CFP is reached. This is reflected by the timer start event of the process at the *Journal PC Chair*. Then, it waits for submissions. The receive activity is authorised until the specified close date, given as (*2021-03-17 T 00:00:00*). When the process receives a research paper, he/she assigns it to a *Reviewer* via the send task assign paper. To avoid delay for the response review process, he/she sends before the deadline date a reminder two times in a period of 15 days between. This is reflected by a non-interrupting boundary time event associated with the receive review task, (*R2/ P15D / 2020-04-29 T 00:00:00*).
- The *Reviewer* process receives a Review Request message to start. The *Reviewer* starts preparing a review, and he/she sends it back to the *Journal PC Chair* when it is ready.
- After the *Journal PC Chair* has received a review, he/she prepares the acceptance/rejection letter or a borderline letter if the paper requires further improvements. Then, he/she attaches the review to the notification letter and sends it to the author at the notification date and time specified in the CFP (*2021-05-16 T 00:00*). The *Journal PC Chair* must conform to the deadline before sending the notification. An intermediate timer event reflects this.

In addition to the problems introduced by the communication as cited in the first example above, this model scenario presents a different need for temporal assurance such as deadlines: Will the collaboration process review be completed on time before the publication's deadlines?. Will the response of the *Reviewer* arrive on time?. Also, it presents the need to use different temporal models such as an activity or a notification that must be done by a predefined date; or the author's response in the case of the second review where it must be done before a period of 15 days starting from a specific date, etc.

2.3.3 BPMN XML Representation

The BPMN structure is described using the BPMN meta-model under the form of class diagrams. BPMN 2.0 is the first release that provides for all the BPMN elements an *Extensible Markup Language (XML)* Schema Definition (XSD) to exchange BPMN 2.0 diagrams between tools, companies, etc. Consequently, BPMN diagrams can be textually represented using an *XML* based notation, saved under *.bpmn*, which is used by several modelling tools, *e.g.*, the Eclipse BPMN Modeller, Camunda Modeller, etc.

In general, the definition of an element in *XML* schema collects all its attributes, visible or invisible, graphically. As an example Listing 2.1 shows the corresponding *XML* fragment of the *Send Task* activity extracted from the collaboration diagram presented in Figure 2.8. The fragment describes two parts: the first (*lines 1-5*) depict the semantics information of the task (id, expression, the incoming and outgoing edges), and the second (*lines 7-8*) depict the localisation of the element in the diagram. We note in this example that the attributes *expression* and *resultsVariables* are invisible on the diagram.

```
1 <bpmn:sendTask id="Task1bn6n5q" name="Make Travel Offer"
2     bpmn:expression="Offre=true" bpmn:resultVariable="Offer">
3   <bpmn:incoming>Flow11b0t0o</bpmn:incoming>
4   <bpmn:outgoing>Flow14ugxng</bpmn:outgoing>
5 </bpmn:sendTask>
6 <bpmndi:BPMNShape id="SendTask0bw2qnzdi" bpmnElement="Task1bn6n5q">
7   <dc:Bounds x="340" y="500" width="100" height="80" />
8 </bpmndi:BPMNShape>
```

Listing 2.1: *Make Travel Offer XML Fragment using Camunda Modeller*

In our work, we use the BPMN *XML* representation for parsing the diagrams and generating the

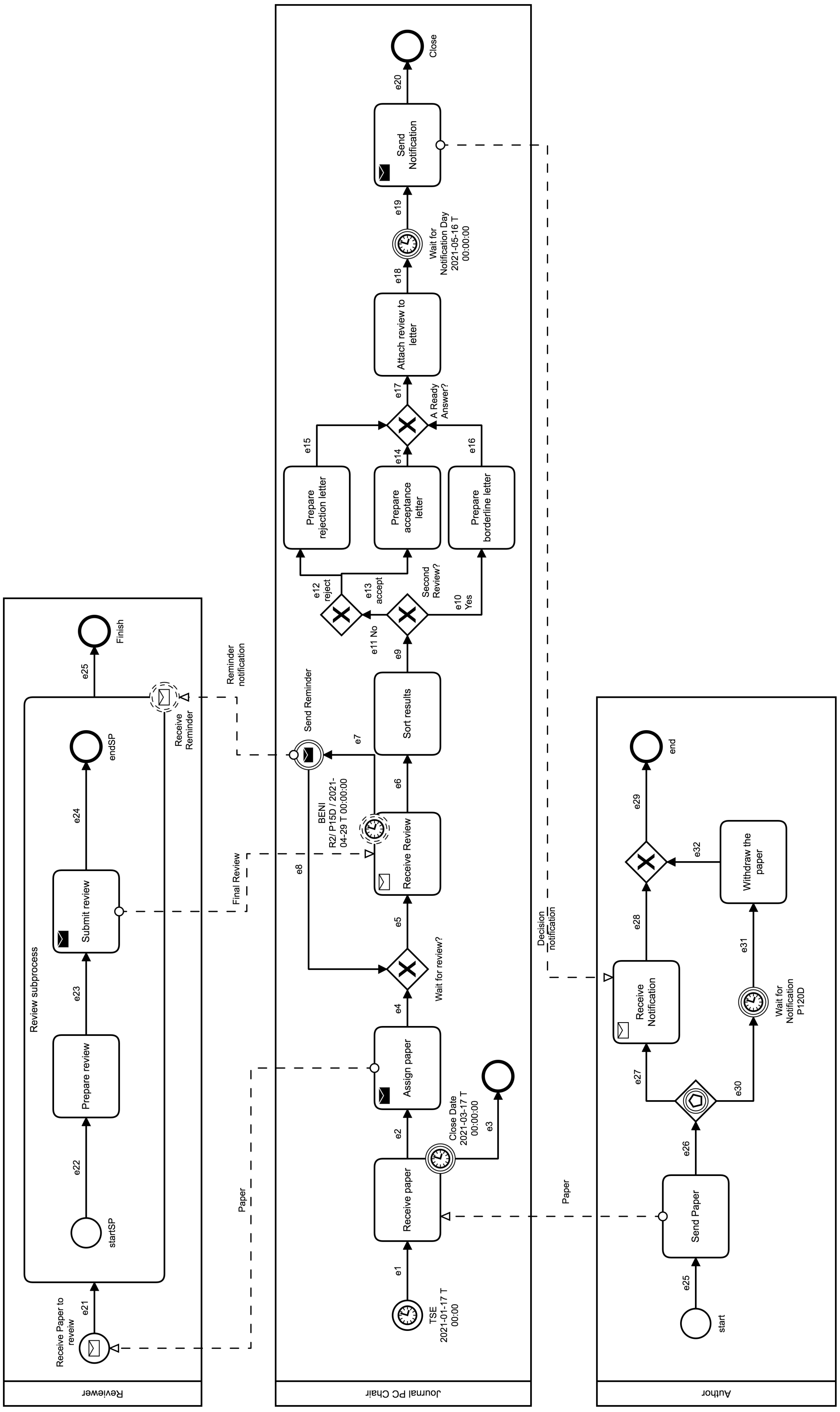


Figure 2.9: *Paper Reviewing Case Study.*

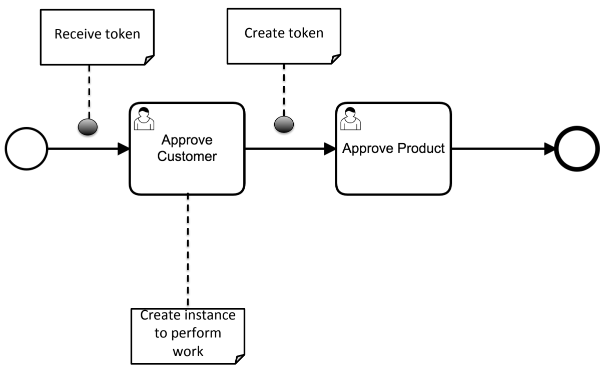


Figure 2.10: *Approver Order Process. (extended from source figure [24]).*

corresponding semantics and in the animation of it.

2.3.4 BPMN Operational Semantics

BPMN 2.0 is more than a drawing language. It supports the process execution under a business process engine. BPMN 2.0 specification extends the scope and capabilities of BPMN 1.x in several areas. Among other improvements, it describes the execution semantics for all BPMN elements. The operational semantics of BPMN defines how the process model elements shall be processed during its execution. When the execution of a business process model starts, a process instance is created. Then, nodes of a process instance are processed and executed according to the order in the control flow of the process model and its underlying operational semantics.

BPMN process is a sequence of activities leading from an initial point of the process instance to some defined endpoints. The question is then *how to start a new process instance?* To answer this question, BPMN provides *Start event* elements (cf. [3, p. 439]), *"For single Start Events, handling consists of starting a new Process instance each time the Event occurs"*. So, *how exactly such a Start event is triggered?*. There are two main alternatives for triggering a *Start event* to create a new process instance: (i) triggers are directly delivered by the environment to the right *Start event* or (ii) triggers are issued into a *Pool* where *Start events* regularly check for appropriate triggers. The (i) represents the configuration of a *None Start event.* While (ii) gives flexibility for the *Start event* node to choose from different triggers, *e.g.*, different messages concurrently available for a *Message Start event.*

The question now is *how the flow nodes and sequence flows processed within the new process instance?*. For answering this question, the BPMN standard uses the key concept of *token* for the description of the execution semantics of the elements. BPMN states that *"a token is a theoretical concept that is used as an aid to define the behaviour of a process that has been performed."* [3, p. 27]. In general, the token traverses the sequence edges, and an end event eventually consumes it. When a start event creates a new process instance, a new token of this new instance is placed on each outgoing sequence flow of the start event. The target nodes of these sequence flows will be enabled when they have received the necessary number of tokens. When the target node is enabled, it can fire and start to work and pass tokens on. When it has fired and completed its actions, it will again produce tokens for the respective instance on its outgoing sequence flows.

Figure 2.10 highlights the general concept of a token traversing the diagram according to the standard. *e.g.*, the activity *Approve Customer* is enabled by a token. If its *startQuantity* attribute is 1 (the default value), then in one step, it will consume the token. Then, after completion of its actual work, assuming its *completionQuantity* attribute is also 1, it will produce one token on its outgoing sequence flows. Therefore, the next activity, *Approve Product*, will be executed in the same manner.

Activities in BPMN represent the set of works that the model will perform. Activities have their instances. The instantiation for an *Activity* is not similar to the *Process.* The common behaviour of all activities is that their instantiation relies on their incoming sequence flows. According to the BPMN, a *subprocess* in a normal flow *"is instantiated when it is reached by a sequence flow token"*[3, p. 430]. It

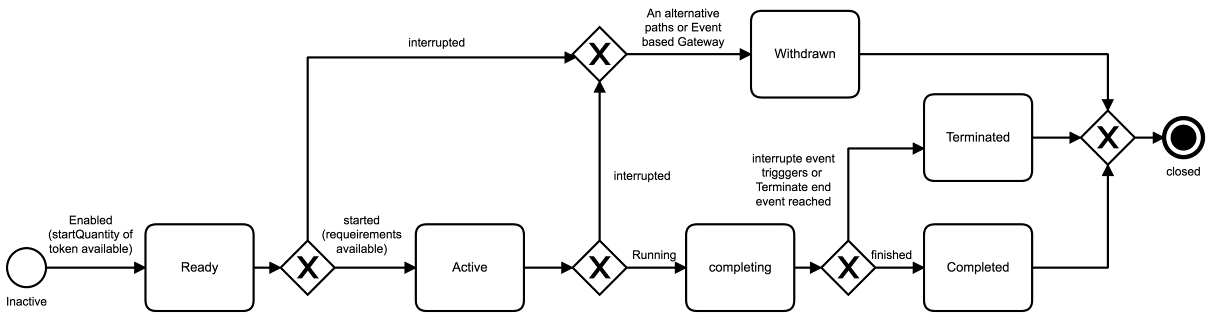

interrupted
An alternative paths or Event based Gateway
Withdrawn
Terminated
closed
interrupted
interrupte event trigggers or Terminate end event reached
Enabled (startQuantity of token available)
Ready
started (requeirements available)
Active
Running
completing
finished
Completed
Inactive

Figure 2.11: *Refined Activity Lifecycle. (from source figure [3])*

has *"a unique empty start event"*, *i.e.*, a *Start event* with a *None* trigger. This *Start event* does not wait for a trigger from outside, but it will be activated after the *subprocess* instantiation. So, contrary to a *Start event* of a process, the start event of the *subprocess* does not create a new process instance as such an instance already exists, but only a new activity instance is created.

When the *Activity* is instantiated, it is executed according to the lifecycle depicted in Figure 2.11. The presented lifecycle model is simplified (cf. Figure 13.2, [3, p. 428]); it ignores compensating, compensated, failing, failed states (as this thesis does not support compensation and error events). In general, when a process instance starts, all its activity nodes are in an inactive state. Activities have to move through a lifecycle, starting from the state *Ready* after being enabled by a number, *startQuantity*, of tokens. If the *Activity* is in *Ready* state, it checks whether input data are available (if required) to move to *Active* state (a state where intended work is performed). An *Activity* in *Ready* or *Active* state can be *Withdrawn* from being able to complete in the context of a race condition. This situation occurs (i) for tasks that are followed an event-based exclusive gateway element, and one of its followed elements (task or event) completes. This causes all other tasks to be withdrawn, or (ii) for alternative paths choices as tasks after an *Exclusive gateway.* Afterwards, during *Completing*, resources can be released, and a clean-up can be made. Moreover, if an activity is in an active or completing state, an *Interrupting event* may be triggered and abort the *Activity.* This leading to *terminate* it. But, if the *Activity* is normally completed, the activity state moves from *Completing* to *Completed* state. Note that the two states *Completed* and *Terminated* are final states. The activity state can be set to *Terminated* from being able to complete in the context of (i) a *Terminate end* event is reached within a process, *"Termination indicates that all Activities in the Process or Activity should be immediately ended."* [3, p. 235] or (ii) an *Interrupting boundary intermediate event* is triggered (*e.g.*, timer events, message events, etc.).

There are two exceptional cases when the *Interrupting event* is attached to a *subprocess.* If the *Interrupting event* has occurred during the execution of the *subprocess*, it ignores it if it is in a final lifecycle state; otherwise, all running, active or ready elements within the *subprocess* are interrupted (*i.e.*, all tokens are deleted). Then, a new life cycle state is set. Indeed, if the *Activity* is nested (*i.e.*, a subprocess activity contains subprocess activities, etc.), all nested *Activities* are interrupted recursively.

We note that BPMN standard states: *"All nested Activities that are not in Ready, Active or a final state (Completed, Compensated, Failed, etc.) and non-interrupting Event subprocesses are terminated"* [3, P. 429]. *I.e.*, all nested activities that are not in *Ready*, *Active* or *Completed* state are terminated. However, the fact that instances in *Ready* or *Active* states should not be interrupted is inconsistent with the diagram of an activity as lifecycle [3, p.428]. For that, we consider above only nested activities in final states.

BPMN defines *Gateways* to control the execution ordering of the activities within a process instance. *Gateways* are used to express splits and/or merges in the control flow of a process. Their semantics is expected to be instantaneous (when they are triggered). As we mentioned before, there are five different gateway types in the BPMN standard while in this thesis we are interested by four: *Parallel*, *Exclusive*, *Event-based*, and *Inclusive* types (see Figure 2.6). The *Exclusive* and the *Event-based* gateways will be enabled when only one of its incoming sequence flows is enabled. Then, they will decide on which outgoing sequence flows are to be taken. The decision for the exclusive gateway depends on data evaluation, while the event-based gateway decision depends on the event triggers of the target elements. The parallel gateway will wait until all its incoming sequence flows are enabled (*i.e.*, each incoming flow has at least one token). Then, it will be fired, and all its outgoing sequence flows will receive a token (*i.e.*, there are no conditions for choosing paths). The *Inclusive* gateway may synchronise a subset, or all its incoming

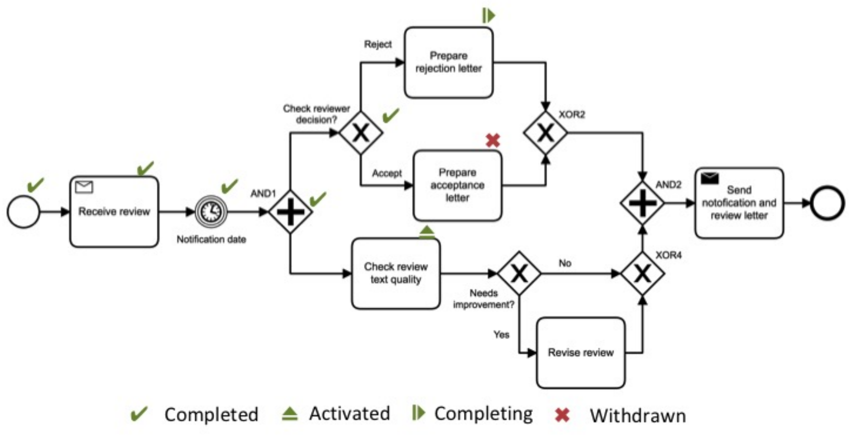


Figure 2.12: *Process Instance of PC Chair Notification Process.*

sequence flows depending on a set of rules and chooses one or more of its outgoing sequence flows, based on data decision to produce tokens there. The semantics of this gateway is relatively complex, and it will be detailed in Chapter 4.

During the process instance execution, the process control flow is influenced by the event occurrences. Whereas activities represent units of work that have a duration, the events are used to model something that happens instantaneously. BPMN standard distinguishes between the Start event that signals how process instances start (tokens are created), the *End event* that signals when process instances are completed (tokens are destroyed), and the *Intermediate* event (within a control flow). The *Intermediate* events cannot retain tokens. Therefore, the BPMN standard specifies that a token remains in the incoming sequence flow of an intermediate event until the event occurs. Once the event occurs, the token traverses the event instantaneously.

Catching events specified where the control flow within a process waits for something to happen, *e.g.*, the arrival of a message, the occurrence of a signal, or temporal deadline has been reached. The enabling of the event depends on (i) the presence of a token on one of its incoming edges or on the activity to which the boundary event belongs (if it is of boundary type) and (ii) the availability of triggers corresponding to the event trigger type. The firing of a catching event generates a token on its outgoing edges. The firing of the *Boundary* event may either start an exception flow for the running instance of the activity attached to (alternative to the normal flow) or interrupt the running instance and produce a token on their outgoing or, if not exist, on the outgoing edge of the activity attached to.

Throwing events are flow nodes that throw an event trigger. They can be either *Intermediate* or *End* events. All throwing events can fire when at least one token is available on at least one of their incoming sequence flows. If *Throwing* events is an intermediate, it fires by throwing triggers and passing tokens to its outgoing sequence flows. In contrast, the *End* events consume a token without producing a new token. *End* events finished the flow of an instance. A *Process* instance completes when an end event has consumed all its tokens, and no activity is left active. Note that the *Process* ends abnormally when a trigger of the *End* event is of a *Terminate* type. The *End* event additionally ensures that all elements within the process instance are terminated (*i.e.*, drops all tokens of the process instance).

Example 2.3.1. Process Instance.
Figure 2.12 depicts a running process instance of a PC chair notification process model. The execution history state of this process instance is as follows:

- The *Start* event instantiated the process and completed;
- The task *Receive review* executed and completed previously;
- An event *Notification date* occurred (completed);

- AND-split gateway *AND1* was executed.
- The activity *Check review text quality* enabled and then activated;
- The XOR-split *Check Reviewer* decision evaluated;
- Finally, the activity *Prepare rejection letter* is currently Running (in a completing state). This implies that the lower branch of the XOR-split *Check Reviewer* is not selected, and the activity *Prepare acceptance letter* is passed to the withdrawn state.

Generally, a process instance is associated with an execution history capturing all events, activities, gateways executed during its execution, referred to in this thesis by an execution trace. The execution trace for this example contains all elements marked as completed (*i.e.*, elements terminates with the withdrawn state are skipped).

2.4 First-Order Logic (FOL)

At the core of every language is the logic that provides the fundamental concepts. Logic is the study of the principles of correct reasoning. The reasoning is a manner to obtain a conclusion from hypotheses. Correct reasoning gives nothing to the truth of the hypothesis; it only guarantees that we can deduce the truth of the conclusion from the truth of the hypothesis. For a long time, logic was associated with First-Order Logic (FOL). FOL is by now a well-understood language with high expressive power and a rich model theory. FOL and its extensions play an important role in many branches of (theoretical) computer science such as (automata theory [25], complexity theory [26], databases [27]). FOL is a small, simple, and expressive language designed for expressing abstractions. It is used to briefly articulate the natural language statements and develop information related to objects very easily. This section presents FOL syntax and semantics briefly. The following notions and notations are fairly standard and can be found in different composition in standard texts about first-order logic, *e.g.*, in [28], [29], [30], or [31].

2.4.1 Syntax of First-Order Formulas

We introduce the syntax of first-order logic (FOL) based on a signature σ, a set containing constant, function, and predicate symbols, each predicate symbol with an arity. First-order formulas over a signature σ are built according to the following syntax elements, two sorts of symbols coexist in FOL:

- Logical symbols, they have a fixed meaning or use in the language
  - *Logical operators:* $\neg$, $\vee$, $\wedge$, $\implies$, and $\iff$, respectively called negation, disjunction, conjunction, implication, equivalence.
  - *Punctuation* "(", ")", opening and closing parenthesis, plus the comma character ",".
  - *Quantifiers:* $\forall, \exists$, respectively called universal and existential quantifier.
  - *Variables:* an infinite set of variables $\mathcal{V}$.
  - *Constants:* an infinite set of constants $\mathcal{C}$.
- Non- logical symbols, they have an application- dependent meaning or use:
  - *Functions:* for each natural number $n > 0$, an infinite set $\mathcal{F}_n$ (n-ary function).
  - *Predicates:* for each natural number $n > 0$, an infinite set $\mathcal{P}_n$ (n-ary predicate).

Definition 2.4.1. (Signature) A *vocabulary* $\sigma = (\mathcal{V}, \mathcal{O}, ar)$ comprises a non-empty set of variables $\mathcal{V}$, a non-logical operator symbols $\mathcal{O} = \mathcal{P} \cup \mathcal{F}$ contains a countable set $\mathcal{P}$ of predicate symbols and a countable set $\mathcal{F}$ of function symbols, and function ar associates to each symbol in $\mathcal{O}$ a non-negative integer.

Note: A function symbol of arity zero is called constant symbol. A predicate symbol of arity zero is called proposition.
Based on the vocabulary defined above, we now give the rules to build what we call a strict formula. To simplify the definition of formulas, we define the notion of σ-Terms:

Definition 2.4.2. ($\sigma - Term$) Given a certain fixed *vocabulary* σ. A σ-Term is a finite syntactic object:

- any variable $v \in \mathcal{V}$ is a σ-Term;

- if $t_1, ..., t_n$ are σ-Terms, and f is a function symbol $f \in \mathcal{F}$ with n-arity $ar(f) = \{n | n > 0\}$, the expression $f(t_1, ..., t_n)$ is also a σ-Term;

Definition 2.4.3. (σ-Formula) σ-Formula can be seen as a list of symbols from the vocabulary (operators, parenthesis, variables, and constants), it is at least satisfying one of these constraints:

1. It can be an atomic σ-Formula:
   - $P(t_1, \ldots t_n)$ is a formula, where the t_i are σ-Terms, P is a predicate symbol from $\mathcal{P}$, of arity n, with $n > 0$;
   - $t_1 = t_2$ is a formula, where t_1 and t_2 are terms;
2. Or a non-atomic σ-Formula:
   - a σ-formula is either: a σ-formula φ, a σ-formula ψ, a negated σ-formula $\neg\varphi$, a conjunction $\varphi \wedge \psi$, a disjunction $\varphi \vee \psi$, an implication $\varphi \implies \psi$, or an equivalence $\varphi \iff \psi$;
   - a quantified σ-formula of the form $\forall x.\psi$ or $\exists y.\psi$ where φ, ψ are σ-formulas and $x, y \in \mathcal{V}$ are first-order variables;

Example: Consider a signature with a constant symbol 1, binary function symbol +, and a binary relation symbol <. Then $x + 1$ is a term and $\forall x(x < (y + 1))$ is a formula.

Definition 2.4.4. (Bound and Free Variables) A variable v occurs *freely* in a formula φ if the formula contains an occurrence of v that is not in the scope of any quantifier $\mathcal{Q}$ such that $\mathcal{Q} \in \{\exists, \forall\}$. An occurrence of a variable v in a formula φ is *bound*, if it lies within the scope of some quantifier $\mathcal{Q}$ Note that different occurrences of the same variable in a given formula can be both bound and free, *e.g.*, variable x occurs both bound and free in the formula $P(x) \wedge \exists x P(x)$.

Definition 2.4.5. (Close Formula) A formula is *closed* if it does not contain any free variables.

Definition 2.4.6. (Sub-Formula) If a σ-formula φ occurs as part of another σ-formula ψ, then φ is called a subformula of ψ.

Definition 2.4.7. (Operators priority order) The priority of the different operators is given in decreasing priority order: negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\implies$), and equivalence ($\iff$). For equal priority, the left operator has a higher priority, except for the implication which is right associative (i.e., $a \implies b \implies c = (a \implies (b \implies c))$).

Important notes: (i) To avoid too many parentheses in formulas and simplify their readability, some parentheses may be deleted to form *priority formulas*. In the rest of this manuscript, we follow the convention of Operators priority order mentioned above to define the formulas in some abbreviation form (*e.g.*, the abbreviation of the formula $((a \wedge b) \vee c)$ is $a \wedge b \vee c$). (ii) In general, we speak of terms, formulas, and atoms when σ is not important or clear from the current context.

Definition 2.4.8. (Literal) A literal is a variable or its negation. For a given variable x, the positive literal is represented by x, and the negative one by $\neg x$.

Some formulas have remarkable structural properties. We say that they are in a *normal form*. To introduce the different definitions of normal forms we are interested in, we first need to introduce the concepts of *cube*, and *clause*.

Definition 2.4.9. (Cube) A cube is a conjunction of literals.

Definition 2.4.10. (Clause) A clause is a disjunction of literals. A unit clause is a clause containing precisely one literal.

There exist many normal forms, but we present here only two forms we are interested in: the conjunctive normal form and the disjunctive normal form.

Definition 2.4.11. (Conjunctive Normal form) A conjunctive normal form (CNF) formula is a finite conjunction of two or more clauses $\varphi = \bigwedge_{i=1}^{n} \psi_i$, where $\psi = \bigvee_{j=1}^{k} l_j$ is a clause, and $n \in \mathbb{N}$. A CNF formula is possibly preceded by a quantifier prefix. A formula in CNF is *Horn* if every clause contains at most one non-negated literal. It is *Krom* if every clause contains at most two literals.

Definition 2.4.12. (Disjunctive Normal form) A disjunctive normal form (DNF) formula is a finite disjunction of two or more cubes $\varphi = \bigvee_{i=1}^{n} \psi_i$, where $\psi = \bigwedge_{j=1}^{k} l_j$ is cube, and $n \in \mathbb{N}$.

Example: Given the variables a, b, c ,d, e, we define the formula $\varphi_1 = (a \vee \neg b \vee \neg c) \wedge (\neg d \vee e \vee f)$, φ as a CNF formula with clauses $(a \vee \neg b \vee \neg c), (\neg d \vee e \vee f)$. The formula $\varphi_2 = (a \wedge \neg b \wedge \neg c) \vee (\neg d \wedge e \wedge f)$ is a DNF formula where $(a \wedge \neg b \wedge \neg c)$ and $(\neg d \wedge e \wedge f)$ are cubes.

2.4.2 Semantics of First-Order Logic Formulas

The following section aims to present the interpretation of the logic formulas presented for the given structures above.

Definition 2.4.13. (σ-structure) Let $\sigma = (\mathcal{V}, \mathcal{O}, ar)$ be a vocabulary, a σ-structure (or assignment, or a model) $\mathcal{A} = (\mathcal{U}_\mathcal{A}, \alpha, \mathcal{I})$ consists of:

- $\mathcal{U}_\mathcal{A}$, a non-empty set called universe of the structure (domain). The domain $\mathcal{U}_\mathcal{A}$ contains two fixed truth values, true and false, noted here by $\mathbb{B} = \{\top, \bot\}$;
- $\mathcal{I}$, an interpretation function that assigns elements of the FOL language to objects in the domain of the interpretation $\mathcal{U}_\mathcal{A}$. More precisely:
  - $\mathcal{I}$ interprets any n-ary predicate symbol $p \in \mathcal{P}$ of arity $n \in \mathbb{N}$ by a set of n-tuples over $\mathcal{U}_\mathcal{A}$ domain, such that: $\mathcal{I}(p) : {\mathcal{U}_\mathcal{A}}^n \to \mathbb{B}$. We note that the set may be empty;
  - $\mathcal{I}$ interprets any n-ary function symbol $f \in \mathcal{F}$ of arity $n \in \mathbb{N}$, $\mathcal{I}(f)$ by an n-ary function over a universe domain $\mathcal{U}_\mathcal{A}$, such that :$\mathcal{I}(f) : {\mathcal{U}_\mathcal{A}}^n \to \mathbb{B}^*$;
- $\alpha : \mathcal{V} \cup \mathcal{C} \to \mathcal{U}_\mathcal{A}$, a valuation function that assigns:
  - for each constant symbol $c \in \mathcal{C}$, an element $c_\mathcal{A}$ of $\mathcal{U}_\mathcal{A}$;
  - for each variable $x \in \mathcal{V}$, an element $x_\mathcal{A}$ of $\mathcal{U}_\mathcal{A}$;

Example 2.4.1. Example $(\mathbb{N}, <, 0)$ denotes the structure with universe $\mathbb{N}$, binary relation $<$ (understood as the usual order on $\mathbb{N}$), and constant 0. Note though that this convention does not specify which values are assigned to variables.

Definition 2.4.14. (Evaluation of the Formula) The value of the formula depends on the value of its terms. Let t be a term, $\mathcal{A}$ a $\sigma - structure$, and α a variable assignment over the universe domain $\mathcal{U}_\mathcal{A}$. We denote the evaluation of t under α and $\mathcal{A}$ by $[t]_\alpha$. We have:

- for a variable $v \in \mathcal{V}$, $[v]_\alpha = \alpha(x)$;
- for a constant $c \in \mathcal{C}$, $[c]_\alpha = c_\mathcal{A}$, with $c_\mathcal{A} \in \mathcal{U}_\mathcal{A}$. eg., $x \in \{\top, \bot\}$, $[\top]_\alpha = 1$, $[\bot]_\alpha = 0$;
- for $f(t_1, ..., t_n)$ a term defined for $n > 0$, f is an $n - ary$ function and $t_1, ..., t_n$ are terms, then $[f(t_1, ..., t_n)]_\alpha = \mathcal{I}(f)([t_1]_\alpha, ..., [t_n]_\alpha)$

Based on the value it can take the formula, it can be satisfiable or unsatisfiable.

Definition 2.4.15. (Satisfaction relation) Let G and F be formulas, x a variable, $\mathcal{A}$ a σ-structure, α a variable assignment. We define the satisfaction relation $\mathcal{A}, \alpha \models F$ ($\mathcal{A}, \alpha$ satisfies F or $\mathcal{A}, \alpha$ models F) by induction over the structure of formulas we have:

- $\mathcal{A}, \alpha \models \mathcal{P}(t1, \ldots, t_n)$ if $([t1]_\alpha, \ldots, [t_n]_\alpha) \in \mathcal{I}(\mathcal{P})$.
- $\mathcal{A}, \alpha \models (F \wedge G)$ if and only if $\mathcal{A}, \alpha \models F$ and $\mathcal{A}, \alpha \models G$.
- $\mathcal{A}, \alpha \models (F \vee G)$ if and only if $\mathcal{A}, \alpha \models F$ or $\mathcal{A}, \alpha \models G$.
- $\mathcal{A}, \alpha \models \neg F$ if and only if $\mathcal{A}, \alpha \not\models F$.
- $\mathcal{A}, \alpha \models \exists x\ F$ if and only if there exists $a \in \mathcal{U}_\mathcal{A}$ such that $\mathcal{A}, \alpha[x \mapsto a] \models F$.
- $\mathcal{A}, \alpha \models \forall x\ F$ if and only if $\mathcal{A}, \alpha[x \mapsto a] \models F$ for all $a \in \mathcal{U}_\mathcal{A}$.
- $\mathcal{A}, \alpha \models t1 = t2$ if and only if $[t1]_\alpha = [t2]_\alpha$.

Notation. We note that if there is no confusion, the expression of the form $\mathcal{A}, [v_1 \mapsto a_1, \dots, v_n \mapsto a_n] \models \varphi(v_1, \dots, v_n)$ by $\mathcal{A} \models \varphi(v_1, \dots, v_n)$. In such case, $\mathcal{A} \models \varphi$ if $\mathcal{A}, \alpha \models \varphi$ holds for every variable assignment α over $\mathcal{A}$ universe domain.

Definition 2.4.16. (Model/Counter-Model of a Formula) For a given formula φ, a σ-structure $\mathcal{A}$, if $\mathcal{A} \models \varphi$, we say that the formula φ is satisfiable and the σ-structure $\mathcal{A}$ is a model of φ. Otherwise, φ is unsatisfiable and $\mathcal{A}$ is a counter-model.

Definition 2.4.17. (Tautology) F is called valid if $\mathcal{A} \models F$ for *every* σ-structure $\mathcal{A}$. Given a set of formulas S, we write $S \models F$ to mean that every σ-structure $\mathcal{A}$ that satisfies S also satisfies F. The same relations exist in propositional logic, *e.g.*, F is un-satisfiable if and only if $\neg F$ is valid.

It is clear from the latter that the formula's evaluation depends on its structure and the values of its variables. Further, the value of a formula depends on all the assignments of its variables. Depending on the value taken by the formula for a particular assignment, we call this assignment model or counter-model of the formula. Table 2.1 presents the different possible values of a given formula φ, regarding different logical operators meaning of the vocabulary. Given $\mathcal{V} = \{x, y\}$ a set of variables, α an assignment function that maps each variable to a value in the set $\mathbb{B} = \{\top, \bot\}$. The evaluation of the formula is then obtained by replacing the variables with their values.

Table 2.1: *Example of Truth Table Evaluation of a Set of Formulas.*

$\mathcal{V}$		φ							
x	y	$\neg x$	$x \vee y$	$x \wedge y$	$x \implies y$	$x \iff y$	$x \wedge \neg y$	$\neg(x \implies y)$	$(x \implies y) \iff (\neg y \implies \neg x)$
$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$
$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\top$

2.5 Verification Methods

The verification of a system is the act of determining or refuting the system's accuracy with respect to its documentation or its formal specifications. There are different methods to analyse the efficiency of the system using formal or informal methods. Formal methods consist in proving the correctness of the system using mathematical tools. Informal ones use static and/or dynamic analysis. In the following, we present a set of verification approaches, emphasising their advantages and disadvantages.

2.5.1 Test

To verify the correct functioning of programs, the most used method in industry is testing. It is a dynamic approach that requires running the program on a set of inputs to ensure that the produced results are consistent. The program's validation is based on the association between the tested inputs and the expected outputs of the program. These tests can be carried out on the whole program (often referred to as *Integration and Validation Test*) or on isolated features (*unit test*).

The main advantage of the test is its sureness. The achieved result is precisely the one produced by the program since it is obtained by running it. Consequently, if the analysis indicates that a result is not as expected, the verdict is reliable. However, the test is not exhaustive, it can only verify a limited number of inputs to a given program, and the concrete inputs can be large or even endless. Accordingly, if the tests find no defects, unless they are carried out exhaustively for all possible program inputs, it does not guarantee its safety.

There are various metrics to determine the quality of a program's coverage by testing [32]. These can be exploited, in combination with static analyses, to automatically generate many tests that can improve the quality of coverage and, therefore, the confidence that one can have in the program. As opposed to dynamic analyses, static analyses are performed without running the code.

In general, the test method eliminates a reasonable number of errors with moderate cost. It allows to exhibit on the performed tests set the absence of a certain number of behaviours identified as problematic or the presence of behaviours identified as intended, but not for all inputs and the behaviours defined by the general specification. For this, it is necessary to turn to formal methods.

2.5.2 Abstract Interpretation

Abstract interpretation [33] is a static analysis technique [34], reasons on all behaviours that could arise during the execution of a program to determine if they are included in the set of acceptable ones. This analysis is generally undecidable [35] and uses an approximation based on the structure theory and the fixed-point calculus. Users analyse the behaviour on all possible inputs without executing the instructions for each of them. The principal is to solve a set of equations determined from the program, expressing its semantics. The limit of this method relates to the ability of tools to solve these equations. If there is no automatic decision procedure for their resolution, the results are still very imprecise.

2.5.3 Theorem Proving

Theorem proving consists of expressing the system and the desired properties as axioms and inference rules, defining a theorem. The properties are then verified on the system using a *theorem prover*, *e.g.*, Coq [36] or Isabelle [37], HOL Proof Assistants [38]. The calculi used for these proofs are introduced by Hoare Logics [39]. Hoare's logics allow reasoning about programs and their properties. Hoare's triplet has the form $\{P\}\ c\ \{Q\}$ where P and Q are logical assertions and c a sequence of instructions. It indicates that from a state where the program respects the property P, the sequence of instructions c leads to a new state where Q is respected. In Hoare's logic, it is possible to reason in terms of total or partial correctness. In total correctness, Hoare's logic is based on inference systems on the triplet mentioned above. Each program instruction and the successive composition of this instruction results in a triplet in the derivation tree. Therefore, it is desirable to automate this inference as much as possible. However, for some instructions, the calculation cannot be automated. This is particularly the case of loops for which it is necessary to indicate invariants that must be respected.

The main advantage of the theorem proving method is its usability in the case of infinite systems. However, tools using this method commonly have the disadvantage of not being completely automatic. In the case of automatic solvers, it is possible to guide them by adding intermediate assertions in the source code or additional lemmas in the knowledge base, but this requires some tools expertise to determine why the proof does not pass. It is also possible to carry out the proofs by using interactive provers. However, the proofs produced by the generators are often complex to read because of too strong or too weak simplifications made by the *Verification Condition Generator (VCGen)*, as well as to the encoding of semantics for the target provers. This issue represents the main barrier preventing its wider adoption by the industry.

2.5.4 Model-Checking

Model-checking [10] is a technique for verifying information or electronic systems represented by a model. The system is presented in a finite state automaton (*i.e.*, labelled transition system graph) from which the desired properties can be checked. Figure 2.13 shows the principle of the model checking technique. Given the model specification and the properties as inputs to the model checker, it solves the property satisfiability question with respect to the model by exploring the state space exhaustively. If the property is not verified, it generates paths invalidating the property. In this case, the user must analyse these paths to determine whether the system's error is due to its model representation or the system itself is indeed at fault.

The advantage of model-checking is the automatic nature of the verification procedure and the generation of counterexamples. Therefore, the user's work is reduced to the system's formal modelling and the properties definition. However, one of the model-checking disadvantages concerns the problem of the combinatorial explosion of state space. This problem is due to the exponential increase in the size of the system state space according to processes' number and the components' number per process [40]. It is possible to control the combinatorial explosion by representing states symbolically and not concretely. In the case of models where the number of states is huge, this representation may not be sufficient to allow verification. In particular, this is the case when the number of states is infinite. In this context, abstraction methods are used to build an approximate model of the verified system using only a finite number of states. This type of analysis is implemented by several tools in the literature(*i.e.*, SPIN [41], Cubicle [42], UPPAAL [43], TLA$^+$ [17], etc.).

2.6 Verification Languages and Frameworks

There is a wide range of formal specification languages based on various logics and other formalisms. A subset of these languages, in which we are interested, is the so-called *model based formal specification*

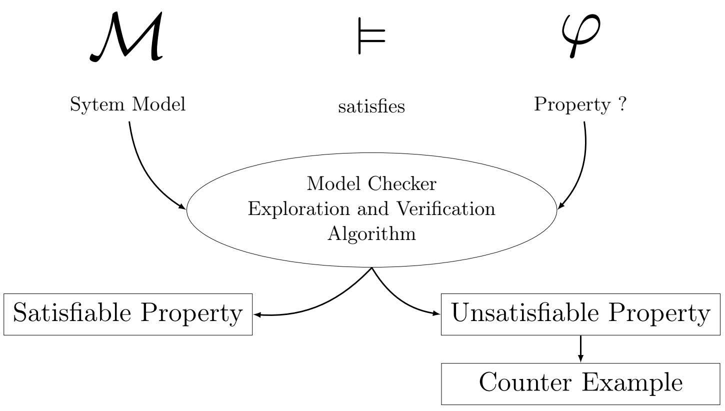


Figure 2.13: *Model Checking Method.*

$\langle Formula\rangle \triangleq$	$\langle Predicate\rangle \mid \Box[\langle Action\rangle_{\langle Statefunction\rangle}] \mid \neg\langle Formula\rangle$ $\mid \Box[\langle Formula\rangle] \mid \langle Formula\rangle \wedge \langle Formula\rangle$
$\langle Action\rangle \triangleq$	Boolean-valued expression containing constants symbols and variables with primed and non-primed variables
$\langle Predicate\rangle \triangleq$	$\langle Formula\rangle$with no primed variables $\mid$ ENABLED$\langle action\rangle$
$\langle Statefunction\rangle \triangle$	nonboolean expression containing constants symbols and variables

Figure 2.14: *An Excerpt of TLA Syntax (figure taken from [17]).*

languages. Their approach for specification consists of describing systems by building mathematical models. Traditionally, the system specification is expressed as a system state model. This state model is constructed using well-understood mathematical entities (*e.g.*, sets and functions). Among the existing languages, we choose languages based on model checking verification methods: *Temporal Logic of Actions (TLA$^+$) and the Lightweight formal methods (Alloy).* In the following sections, we give a brief overview of them.

2.6.1 TLA Logic and TLA$^+$ Language

Temporal Logic of Actions (TLA$^+$) [17] is a formal specification language for specifying and automatically verifying concurrent and distributed systems. TLA$^+$ is based on the combination of TLA (Temporal logic of Actions) and ZF (Zermelo-Fraenkel) set theories. TLA is parametrised by an underlying first-order language. Its non-temporal fragment is based on a Zermelo-Fraenkel set theory with the axiom of choice for specifying the data structures. TLA$^+$ is a rich language with well-defined syntax and semantics for formal reasoning and is designed to write clear and expressive specifications. In the following, we give a summary of them. Section 2.6.1.1 presents the syntax and the semantics of TLA logic. Section 2.6.1.2 describes the TLA$^+$ language. Section 2.6.1.4 presents the TLA$^+$ model-checker, called TLC (*Temporal Logic Checker*). For a complete presentation of the TLA$^+$ language, we refer the reader to [17].

2.6.1.1 TLA Logic Syntax & Semantics

TLA is a logic proposed by Leslie Lamport [17] to describe concurrent systems. It is an extension of linear temporal logic into a logic of actions.

Figure 2.14 summarises the syntax of TLA. TLA formulas are formed using boolean operators of predicate logic, arithmetic operators, set operators, and LTL operators. The semantics of TLA is based on the concept of state. A state corresponds to a valuation of variables. It is a mapping function from a set of variables Var to a set of values Val. The set of all possible states is the set of all possible valuations of variables, noted St. Lamport denotes by $[[F]]$ the semantic interpretation of a syntactic object F (which can be a predicate, an action or a formula). For example, for a variable v, the semantic interpretation $[[v]]$ is a function that assigns the variable v a value in Val. The value of x in-state s is denoted by $s[[x]]$ (postfix notation used by Lamport). Using these notations, in the following, we will present some basic notions in the TLA language.

State Function. A state function is a general non-Boolean expression, made up of variables and constants operators, that maps each state to a value. Semantically, the interpretation $[[f]]$ of a state function f is a function from a set of states St to the set of values Val.

Predicate. A predicate P is a Boolean expression of variables and constants operators that assigns boolean value to each state. Semantically, it is a function such that $s[[P]]$ assigns the value $true$ or $false$ to a state s.

Action. An action is a boolean expression containing constants, variables and primed variables (noted with ($'$) operator). Unprimed variables refer to variable values in the current state, and primed variables refer to their values in the next state. Thus, the action represents a relation between old states and new states. Semantically, the interpretation $[[A]]$ of an action A is a function which associates to a pair of states (s, t) a Boolean denoted by $s[[A]]t$, with s and t representing respectively current state and next state.

Formula. TLA formulas are built up from actions and predicates using Boolean operators ($\neg$ and $\wedge$ and others that can be derived from these two), quantification over logical variables ($\forall$, $\exists$), the operators $'$ and the unary temporal operator $\Box$ (always) of LTL.

Divers. TLA language provides a set of predefined predicates and formulas. We present them briefly in the following. Let $\mathcal{A}$ be an action, and $vars$ a set of variables used in $\mathcal{A}$.

- ENABLED $\mathcal{A}$, expresses whether the action $\mathcal{A}$ can be executed in a current state s. It is evaluated to true in state s if it is possible to reach some state t from s.
- UNCHANGED $vars$, expresses that the variables $vars$ remain unchanged in the state next, it is an abbreviation of $vars' = vars$.
- $[\mathcal{A}]_{vars}$ is a formula that is evaluated to true if action $\mathcal{A}$ is executable or if the values of the variables of $vars$ remain unchanged.
- $\langle\mathcal{A}\rangle_{vars}$ is a formula that expresses that the variables of $vars$ will change value by executing action $\mathcal{A}$.

Quantifiers in TLA. TLA has existential and universal quantification operators of the logic of predicates. Given a temporal formula φ, a finite set of constants S, the formula:

- $\forall x \in S : \varphi$, is true if and only if φ is true for all the value of x in S.
- $\exists x \in S : \varphi$, is true if and only if φ is true for at least one element x in S.

In addition, TLA defines *unbounded* quantifiers, but since TLC does not support this type of quantifiers, we discard them from our description.

Temporal Formula. It is a Boolean-valued expression that can contain flexible and rigid variables and temporal operators. A *flexible* variable can have a different value in different states of behaviour. A *rigid* variable can have the same value in every state of behaviour. The *behaviour* of the system is an infinite sequence of states. A *state* is an assignment of values to all flexible variables (there are infinitely many flexible variables, but only a finite number of them can occur in any single formula). A behaviour represents a possible history of the observable universe. Terminating executions of a system are characterised by behaviours ending in an infinite sequence of *stuttering steps*. Stuttering steps are steps (*i.e.*, a step is a pair of states) in which no variables in the formula change. A specification of a system should allow stuttering steps since it should allow changes to parts of the universe external to the system. All TLA formulas are invariant under stuttering, meaning that adding or removing stuttering steps does not affect whether or not a behaviour satisfies a temporal formula.

Semantically, the temporal formula is true or false for a behaviour. The temporal operators of TLA can be defined in terms of the primitive operator ("always"), noted $\Box$. Let $\sigma = \langle s_0, s_1, s_2, \dots\rangle$ be a sequence of states, such as $\sigma \in St^{\infty}$ and St^{∞} is the collection of all behaviours.

A TLA temporal formula consists of:

- a predicate P is evaluated to true for a behaviour σ if and only if the initial state of σ is true.

- A formula $\Box P$ is true for a behaviour σ if P is true for every state s_i in σ.
- A formula $\Box[\mathcal{A}]_{vars}$ is true for a behaviour σ if and only if for each pair of successive states (s, t), the action $[\mathcal{A}]_{vars}$ is true.

Let φ be a temporal formula and σ a sequence of states. The Interpretation of $\sigma[[\varphi]]$ is a boolean value that the formula φ assigns to σ. If the interpretation of $\sigma[[\varphi]]$ is true, we say that the sequence σ satisfies or models the temporal formula φ, noted $\sigma \models \varphi$, otherwise, σ un-satisfies φ, $(\sigma \not\models \varphi)$.

To verify that a program satisfies a particular property, we prove that it satisfies a specific temporal formula. We describe in the following the semantics of temporal operators defined from the Boolean operators and the operator always $\Box$. Given φ and ψ temporal formulas, σ a sequence of states:

- *Eventually Operator*, noted by $\Diamond$, is defined in terms of always operator, as the assertion of the negation of a formula does not hold forever: $\Diamond\varphi = \neg\Box\neg\varphi$, states that formulas are not always false. *I.e.*, for a formula $\sigma\Diamond[[\varphi]]$, the sequence σ satisfies φ $(\sigma \models \varphi)$ if and only if φ is true in at least one state of the sequence σ.
- *Always-Eventually*, noted by $\Box\Diamond$, is defined in terms of always and eventually operators, states that formulas are true infinitely often. *i.e.*, for formula $\Box\Diamond\varphi$, the sequence $\sigma \models [[\varphi]]$ if and only if $\Diamond[[\varphi]]$ is true at any instance n in the sequence σ.
- *Eventually-Always*, noted by $\Diamond\Box$, is defined in terms of always and eventually, operators, asserts that a formula eventually holds and it is true in every subsequent state. *I.e.*, $\Diamond\Box[[\varphi]]$ specifies that φ is eventually always true.
- *Temporal Implication*, noted by $\leadsto$, is defined in terms of always and eventually operators, $\varphi \leadsto \psi = \Box(\varphi \implies \Diamond\psi)$, asserts that when a formula being true always-eventually leads to a state where another is true. *I.e.*, when φ is true, then the formula ψ is necessarily true in the future, without requiring that ψ remain true.

Expression of properties in TLA. In TLA, we can express safety, fairness and liveliness properties.

Safety property. Expresses that something bad will never happen. The invariance is a safety property that is expressed in TLA by the always operator $\Box$. Given a predicate P and a specification S. The temporal formula $\Box P$ will be satisfied if the following implication is evaluated to true: $S \implies \Box\ P$.

Fairness property. A system specification is usually a disjunction of actions. Fairness allows specifying that when an action is executable, then it will be executable in the future. Lamport distinguishes two types of fairness, weak and strong fairness.

- *Weak Fairness* states that this action cannot continuously be enabled without being fired. The following formula expresses weak fairness:

$$WF_{vars}(\mathcal{A}) \triangleq \Diamond\Box \text{ ENABLED } \langle\mathcal{A}\rangle_{vars} \implies \Box\Diamond\langle\mathcal{A}\rangle_{vars}$$

- *Strong Fairness* states that an action cannot be infinitely often enabled without being fired (*i.e.*, if an action $\mathcal{A}$ is enabled infinitely often, then it will occur infinitely often as well). The following formula expresses strong fairness:

$$SF_{vars}(\mathcal{A}) \triangleq \Box\Diamond \text{ ENABLED } \langle\mathcal{A}\rangle_{vars} \implies \Box\Diamond\langle\mathcal{A}\rangle_{vars}$$

  Strong Fairness property implies the corresponding *Weak Fairness* property.

Liveness property. expresses that something good will eventually occur. As liveliness is expressed in infinite behaviours, specifications must guarantee progress. Therefore, TLA specifies liveness through weak and strong fairness conditions.

2.6.1.2 TLA$^+$ Language

TLA$^+$ language is a formal specification language that extends TLA logic by structuring it into modules.

Modules. They are used to structure complex specifications. Module declaration starts with the keyword *MODULE*. Each TLA$^+$ module contains a set of constants declaration specified by the keyword *CONSTANTS*, a set of variables declaration specified by the keyword *VARIABLES*, and a set of definitions in term of: *actions, functions, predicates, temporal formulas, and a set of properties to check* using the operator form ($Op(p_1, \ldots, p_n) \triangleq expr$). A module can extend other modules, importing all their declarations and definitions using the keyword *EXTENDS*.

Expression. It relies on standard first-order logic, set operators, and several arithmetic modules.

Set. TLA$^+$ is a set-theoretic language where every expression includes formulas, functions, numbers, etc., denotes a set.

Table 2.2 shows a summary of TLA$^+$ operators. An overview of the syntax TLA$^+$ describes the standard constructs and operators of the language is as follows.

Logic		
	$TRUE\ FALSE\ \wedge\ \vee\ \neg\ \implies\ \equiv$	
Sets		
	$\neq\ \in\ \notin\ \cap\ \cup\ \subset\ \setminus$	[Set operators]
	$\{e_1, \ldots, e_n\}$	[Set consisting of the elements e_i]
	$x \in S : p$	[Set of elements x in S satisfying p]
	$e : x \in S$	[Set of elements e such that x is in S]
	SUBSET S	[Set of subsets of S]
	UNION S	[Union of all the elements of S]
Functions		
	$f[e]$	[Application of the function f to an element e]
	DOMAIN f	[Domain of function f]
	$[x\ \in\ S \mapsto e]$	[Function f such that $f[x] = e$ for $x \in S$]
	$[S \rightarrow T]$	[Set of functions f with $f[x] \in T$ for all $x \in S$]
	$[f$ EXCEPT!$[e1] = e2]$	[Function f' equal to f such that $f'[e1] = e2$]
	$[f$ EXCEPT!$[e1] \in S]$	[Set of functions f' equal to f such that $f'[e] \in S$]
Records		
	$e.h$	[The h field of the e record]
	$[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n\ ]$	[The record whose field e is equal to e_i]
	$[h_1 : S_1, \ldots, h_n : S_n\ ]$	[Set of all records such that each field h_i is in the set S_i]
	$[r$EXCEPT!$.h = e]$	[Record r' equal to r such that $r'.h = e$]
	$\{[r$EXCEPT!$.h]2S]\}$	[Set of records r equal to r such that $r.h \in S$]
Sequences		
	$e[i]$	[The i^{th} element of the sequence e]
	$\langle e_1, \ldots, e_n \rangle$	[The n-tuple whose i^{th} element is e_i]
	$S_1 \times \cdots \times S_n$	[Set of all n-tuples such that the i^{th} element is in the set S_i]
Chains		
	$c_1, \ldots, c_2$	[A sequence of characters n characters]
	STRING	[The set of all character sequences]
Conditional Constructs		
	IF p THEN e_1 ELSE e_2	[Equal to e_1 if p is true, otherwise e_2]
	CASE $p_1 \mapsto e_1 \Box \ldots \Box\ p_n \mapsto e_n$	[Equal to e_i if p_i]
Divers		
	LET$d_1 \triangleq e_1 \ldots d_n \triangleq e_n$INe	[Equals to e in the context of definitions $d_1, \ldots, d_n$]
	$p_1 \wedge \cdots \wedge p_n$	[Conjunction of $p_1 \ldots p_n$]
	$p_1 \vee \cdots \vee p_n$	[Disjunction of $p_1 \ldots p_n$]

Table 2.2: *Summary of TLA$^+$ Operators.*

Functions. They are primitive objects in TLA$^+$. Each function f in TLA$^+$ has a domain of definition, denoted DOMAIN f. The application of a function f to an expression e is written by $f[e]$. $f[e]$ specifies the value of the function f for an expression e (if e is an element of DOMAIN f). The constructor of the function f is denoted by $[x \in X \mapsto e]$. The expression $[x \in X \mapsto e]$ denotes the function with domain X that maps any $x \in X$ to e (*e.g.*, $x \in Nat \mapsto x + 1$ is the function f such that $f[1] = 2$, $f[3] = 4 \ldots$). TLA$^+$ defines EXCEPT operator. The expression $[f$ EXCEPT $![e_1] = e_2]$ is a function that is equal to the function f except at point e_1, where its value is e_2. The expression $e_1 \rightarrow e_2$ denotes the set of all functions with domain e_1 and co-domain e_2.

Sequences. A sequence (or a tuple) is a TLA$^+$ function which is written under the form of:

- 0-tuple, is the empty sequence, noted $\langle\rangle$. It is the unique function having an empty domain:

$$\langle\rangle \triangleq [x \in \{\} \rightarrowtail \{\}]$$

- n-tuple $\langle e_1, \ldots, e_n\rangle$, is a function whose domain of definition is the the interval of integer numbers $1, \ldots, n$, where $\langle e_1, \ldots, e_n\rangle[i] = e_i$. The i^{th} element of the sequence e, noted $e[i]$ The sequence index starts at 1. The meaning of the n-tuple can be given by the $CASE$ expression as follow:

$$\langle e_1, \ldots, e_n\rangle \triangleq [y \in 1 \ldots n \rightarrow CASE\ (y = 1) \rightarrow e_1 | \ldots |\ CASE\ (y = n) \rightarrow e_n]$$

The set of all n-tuples is noted by $S_1 \times \cdots \times S_n$, such that the i^{th} element is in the set S_i. Several operators on sequences are defined in the standard Sequences module. Table 2.3 presents the main operators on sequences used in this thesis.

$Seq(S)$	[the set of all the sequences built from the elements of S]
$Len(S)$	[the length of the sequence S]
$S \circ T$	[the resulting sequence from the concatenation of S and T]
$Head(S)$	[the first element of S]
$Tail(S)$	[the sequence S omitting the first element $S[1]$]
$Append(S, e)$	[the sequence resulting from the addition of the element e to the sequence S]
$SubSeq(S, m, n)$	[the sequence $\langle S[m], S[m+1], \ldots, S[n]\rangle$]

Table 2.3: *Sequences Module Main Operators.*

Records. An r record is written under the form $h_1 \mapsto e_1, \ldots, h_n \mapsto e_n$ in which each field (or component) h_i is equal to e_i. Access to a field is through the expression $r.h_i$. The set of all records noted by $h_1 : S_1, \ldots, h_n : S_n$ such that each field h_i is in the set S_i. The operator EXCEPT, defined on functions can also be used on records as follows:

- r EXCEPT $!.h = e$, is a record r' equal to r such that $r'.h = e$.
- $\{[r$ EXCEPT $!.h] \in S\}$, is a set of records r' equal to r such that $r.h \in S$.

Note that in TLA$^+$, a record is a function whose definition domain is a finite set of String.

Operators. TLA$^+$ standard grammar includes a set of operators. Different from the functions, operators have not a domain of definition. They may be written under the forms: $Op \triangleq exp$, with Op is an identifier and exp is an expression, or under the form $Op(p_1, \ldots, p_n) \triangleq exp$ with p_i an operator of the form $Op(\_, \ldots, \_)$. $Op(p_1, \ldots, p_n) \triangleq exp$ defines an operator symbol Op such that $Op(e_1, \ldots, e_n)$ equals exp, where each p_i is replaced by e_i.

TLA$^+$ has a set of predefined operators, that are:

- **CHOOSE Operator.** Represents a primitive expression writing as:

$$CHOOSE\ x \in S : P(x)$$

  It denotes an arbitrary fixed value x in a set S such that $P(x)$ is true if x value exists, false otherwise.

- **Boolean Operator.** The Boolean operators:

$$TRUE\ FALSE\ \land\ \lor\ \neg\ \implies\ \equiv$$

- **Set Operators.** TLA$^+$ is a set theory language, has a set of theory operators. The most commonly used basic operators are the empty set $\{\}$, the pairing set $\{exp, exp\}$, the power set SUBSET s, the generalized union UNION s, the union set defined using pairs exp $\cup$ exp , and the intersection exp $\cap$ exp. TLA$^+$ also has the following set constructors:
  - $\{exp_1, \ldots, exp_1\}$, with $n \geq 1$, defines the set of elements e_i.
  - $\{x \in S : p\}$, defines the set of elements x of S satisfying the property p.
  - $\{e : x \in S\}$, defined the set of elements e such that x in S.

Arithmetic. TLA$^+$ has a set of arithmetic expressions which are:

$$e ::= \ldots|0|1|2|\ldots|Int| - e|e + e|e - e|e * e|e\%e|e \div e|e < e$$

Divers. TLA$^+$ language provides some constructors which are:

- **Conditional.** Two conditional constructors, inspired by conditional structures in programming languages :
  - The constructor IF p THEN e_1 ELSE e_2, is equal to e_1 if the predicate p is true and to e_2 when p is false. In the case where there are several nested IF-THEN-ELSE constructors, it is easier to use the CASE constructor.
  - The constructor CASE $p_1 \mapsto e_1\Box \ldots \Box\ p_n \mapsto e_n$, is equal to e_i if p_i true.
- **LET/IN Expression.** It is used to define a local expression. The general form of this constructor is as follows: LET$d_1 \triangleq e_1 \ldots d_n \triangleq e_n$IN$e$, with d_i presents a TLA$^+$ definition. This expression equals to e in the context of definitions $d_1, \ldots, d_n$.

Note that the operator symbols correspond to the standard function and predicate symbols of first-order logic. Thus, TLA$^+$ semantics is an extension to the formal semantics of first-order logic with equality.

2.6.1.3 TLA$^+$ Specification

The dynamic system behaviour is expressed in TLA$^+$ as a transition system, with an initial state predicate and actions to describe the transitions. The TLA$^+$ specification of a system is represented by a single predicate describing the behaviour of the system, called *Spec*. For example, formula. 2.1 shows the definition of the *Spec* predicate. The main part of the TLA$^+$ specification consists of an initial predicate, named *Init* and an action formula, named *Next*, where:

$$Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge Fairness \tag{2.1}$$

- *Init*, is the initial state predicate, a proposition that holds for every state that is a valid initial state for the system.
- $\Box[Next]_{Vars}$, is the next-state relation, which states that every state following the initial state must either satisfy the next-state action Next or be a stuttering state.
- $Vars$ is the set of variables required to specify the system. Variables in TLA have no types and can assume any value.
- $Fairness$ is the fairness condition of the specification.

System properties are specified using LTL. They are expressed using eventually, always-eventually, or eventually always operators, or temporal implication formula described above.

The described *Spec* formula asserts the safety, liveness, and fairness properties due to conjoint fairness conditions to *Spec*.

2.6.1.4 TLC Model Checker

The TLA model checker tool, named TLC [44] is used to check TLA$^+$ specifications. The TLC tool has a distributed implementation and a centralised implementation with different capabilities. TLC tests all possible combinations of actions and reports any detected violations of the system's properties. It performs a breadth-first search to traverse the state graph. It starts by generating all the initial states and adds them to the FIFO queue, then launches threads that repeatedly execute the process described below : (i) pick a state from the FIFO queue and generate all its successor states, (ii) for each successor state; check if it satisfies all the invariant properties and adds it to the end of the FIFO queue; (iii) if some successor does not satisfy some invariant property, report an error and print the corresponding counterexample. The TLA$^+$ toolbox, freely available at [45] contains an editor, a pretty-printer, and the TLC model checker, which enumerates states by interpreting TLA$^+$ specifications.

2.6.2 Alloy Logic & Language

Section 2.6.2.1 presents the Alloy logic. Section 2.6.2.2 presents the semantics of the Alloy language. Section 2.6.2.3 presents its verification tool, called Alloy Analyser.

2.6.2.1 Alloy Logic

Alloy [46], often referred to as *"lightweight formal methods"* [47], is a declarative modelling language based on relational logic (RL), a logic with a clear semantics based on relations. This logic provides a powerful yet simple formalism for interpreting Alloy as modelling constructs. It is based on a First-Order Logic (FOL) enhanced with the transitive closure operation to express complex structural and behavioural constraints [47]. Alloy is very strongly influenced by object-oriented modelling. It allows the user to easily classify objects, and associate properties to objects according to their classification. Figure 2.4 gives a summary of Alloy's operators token from [48] and updated with a subset of operators and definitions [47, 49]. The figure introduces the building blocks that underlie Alloy. Alloy is based on *models*. The structure of each model is built from typed relations and atoms.

Atom. An *atom* is a primitive entity that is *indivisible*, (*i.e.*, it cannot be broken into several sub-parts); *immutable* (*i.e.*, its properties do not change over time), and *uninterpreted* (it has no innate property).

Set. It is a well-defined collection of distinct objects.

Relation. A *relation* is a structure that relates atoms. Mathematically, it is a set of tuples, each tuple consisting of a sequence of atoms. The size of the relation is the number of tuples. Further, The arity of the relation is the arity of the tuples. Thus, a *set* is an unary relation, 1-tuple, a *scalar* is a singleton (size 1) unary relation, binary relation is a relation (size 2). Informally, the term relation is used to mention a relation of arity two or more.

Constants Relation. Alloy provides three relational constants (*univ, none, iden*):

- *none* is the empty relation that contains no tuple,
- *univ* is the universal relation that contains every tuple,
- *iden* is the identity relation that maps each atom to itself.

As these constants are used in expressions with variables, they have types.

Expression. In Alloy, expressions are just like mathematical expressions, constructed by nested applications of operators to variables. All expressions denote relations, so every operator takes one or more relations and yields a relation. Operators fall into two categories: *Set operators* and *Relational operators.*

- *Set operators* includes the standards operators on sets (*union, intersection, the difference, in, and* $=$) writing in ASCII form. The two operators *in* and $=$ are used to compare relations by testing whether the tuples of one relation also belong to another. Given A and B, two relations have the same type. The formula A in B is true when every tuple of A is also a tuple of B. The equality is the containment in both directions, $A = B$ is true if A in B $\wedge$ B in A are true.
- *Relational operators* includes (*product, join, transpose, transitive closure, reflexive transitive closure, domain restriction, and range restriction*) operators.
  - The product $A- > B$ of two relations is a relation resulted by taking every combination of a tuple from A and a tuple from B and concatenating them.
  - The join $A.B$ of relations A and B is a relation resulting from the combination of every tuple in A and a tuple in B, including their join, if it exists. *E.g.*, the join of two tuple $X = (z, x_1, x_n)$, $Y = (y_1, y_n, z)$ is the tuple (X, Y) which starts by the second atom of X, x_1 and ends by the before last atom y_n without the element z. Note that the matched atom is removed from the resulting tuple.
  - The transpose of a relation is its inverse. It takes the mirror image of a relation, forming a

new relation by reversing the order of each tuple. It is only defined for binary relations. *E.g.*, the transpose of the relation $R = \{(a_1, b_1), (a_2, a_2)\}$ is $\tilde{}R = \{(b_1, a_1), (a_2, a_2)\}$

– The reflexive-transitive closure of a relation is the smallest enclosing relation that is transitive and reflexive (that is, includes the identity relation). *E.g.*, the transitive closure of a given relation $R = \{(a_1, a_2), (a_2, a_3), (a_3, a_4)\}$ is $\hat{}R = \{(a_1,a_2), (a_1, a_3), (a_1, a_4), (a_2, a_3), (a_2, a_4), (a_3, a_4)\}$. The reflexive-transitive closure of a relation $R = \{(a_1, a_2), (a_2, a_3), (a_3, a_4)\}$ is ${}^*R = \{(a_1, a_1), (a_2, a_2), (a_3, a_3), (a_4, a_2), (a_1, a_2), (a_1, a_3), (a_1, a_4), (a_2, a_3), (a_2, a_4), (a_3, a_4)\}$, *i.e.*, the set of its identity element plus the set of the transitive closure.

– The domain/range restriction. For any set A and any relation B, the domain restriction relation, $A <: B$, is tuples in B beginning with a member of A. The range restriction relation, $B :> A$, is a relation tuple in B ending with a member of A.

In addition, Alloy provides Operators to present compound (larger) formulas which are are made from smaller formulas by combining them using the standard *Logical operators*, and by quantifying formulas using *Quantifiers* that contain free variables over bindings.

- *Logical operators* includes *negation, conjunction, disjunction, implication, bi-implication and condition* which have the standard definitions (see Table 2.4).
- *Quantifiers* includes five operators defined in Alloy which are *all, some, no, one, lone.* Writing:
  - *all A*, states that for all element in the set A,
  - *some A*, states that there is at least one A in the set,
  - *no A*, states that the set is empty,
  - *one A* states that there is exactly one *A* in the set,
  - *lone A*, states that there is at most one *A* in the set or the set is empty.

  These quantifiers can be applied to variables as well as expressions. Note that the quantified variables are always bounded by an expression. Indeed, if no multiplicity quantifier is used, alloy assumes to be one.

2.6.2.2 Alloy Language

The Alloy language uses the Alloy logic with some other constructs to make modules. The module declaration starts by the keyword *module.* Each Alloy module consists of a number of declared constructs, the gross structural elements of an Alloy model are: *modules*, *signatures*, *fields*, *facts*, *predicates*, *functions*, *assertions*, *commands*, *let expressions.* All those terms will be introduced in the following.

Modules. Alloy models are divided into modules. Each module is declared on a separate file using a declaration. Some models or model fragments are used repeatedly in other models. To make reuse convenient and allow structuring of large models, Alloy enables a model to incorporate the contents of other modules using *import* instructions. The module system of Alloy is a simplified version of the package system of Java. Modules can be arranged in a tree and are given pathnames from the root. The names of the files containing the modules, and their locations in the directory hierarchy, must match the module names. The general module structure is as follows:

module ::= **module** [packageName/] moduleName import... *paragraph* ...
import ::= (**open** | **uses**) packageName [/*]
paragraph ::= signature | fact | assertion | function | run | check | eval

Signatures. Defines a set of atoms representing the basic entities of Alloy. The declaration of a signature is made using the keyword *Sig* followed by the name of the signature, which has the form *Sig S*{...}. Signature declarations may introduce fields. A field represents a relation among signatures, has the form *Sig s* {*f* : *t*} where *f* is a field, *s* and *t* are types, *f* here is a total function, mapping each atom in *s* to exactly one atom in *t*, *e.g.*, *Sig Student* {*class* : *one Class*}. The signature is composed of non-dividable entities called atoms, whose type is given by signatures, and of atom *tuples*, whose type is given by fields. A signature can also introduces *sub-signature* using the keyword *extends*,

Basics			
	$problem :: decl * formula$		
	$decl :: var : typeexpr$		$M : form \rightarrow env \rightarrow Boolean$
Type of Expression			
	$typeexpr :=$		$X : expr \rightarrow env \rightarrow value$
	$type$		$env = (var + type) \rightarrow value$
	$type- > type$		$value = (atom \times \cdots \times atom)+$
	$type => typeexpr$		$atom \rightarrow value$
Expression			
	$expr :=$		
Set Operators			
	expr in expr	[subset]	$M[\text{a in b}]e = X[a]e \subseteq X[b]e$
	expr + expr	[union]	$X[a + b]e \cup X[b]e$
	expr&expr	[intersection]	$X[a + b]e \cap X[b]e$
	expr-expr	[difference]	$X[a - -b]e = X[a]e \setminus X[b]e$
	expr=expr	[equality]	
Relation Operators			
	$R_1- > R_2$	[product]	is the standard cartesian product
	$expr.expr$ or $expr[expr]$	[navigation (join)]	$X[a.b]e = \{(x,z) \mid \exists y.(x,y) \in X[a]e \wedge (y,z) \in X[b]e\}$
	$\tilde{}expr$	[transpose]	$X[\tilde{}a]e = \{\langle x,y\rangle : \langle y,x\rangle \in X[a]e\}$
	$\hat{}expr$	[transitive closure]	$\hat{}r$ is the smallest relation that contains r and is transitive.
	$* expr$	[reflexive transitive closure]	*r is the smallest relation that contains r and is both transitive and reflexive
	$SetD <: Rel$	[domain restriction]	creates a new relation where the domain of R is restricted to D
	$Rel :> SetR$	[range restriction]	creates a new relation where the range of R is restricted to R
Logic Operator			
	$formula :=$		
	$!formula$	[neg]	$M[!F]e = \neg M[F]e$
	$formula\&\&formula$	[conjunction]	$M[F\&\&G]e = M[F]e \wedge M[G]e$
	$formula \mid\mid formula$	[disjunction]	$M[F \mid\mid G]e = M[F]e \vee M[G]e$
	$formula => formula$	[implication]	$M[F => G]e = \neg M[F]e \vee M[G]e$
	$formula <=> formula$	[biimplication]	$M[F <=> G]e = (M[F]e \wedge M[G]e) \vee (\neg M[F]e \wedge \neg M[G]e)$
	formula => formula, formula	[Conditional]	$M[F => G, H]e$ if F then G else H
Quantifiers	noted by Q here		
	all v : type | formula	[universal]	M [all v : t] | $F] = \bigwedge\{M[F](e \oplus v \rightarrowtail \{x\}) \mid x \in e(t)\}$
			F is true for every v in t.
	some v : type | formula	[existential]	M[some v : t | F] = $\bigvee\{M[F](e \oplus v \rightarrowtail \{x\}) \mid x \in e(t)\}$
			F is true for some v in t
	no v : type | formula	[not exist]	M[no v : t | F] =| $\{M[F](e \oplus v \rightarrowtail \{x\}) \mid x \in e(t)\}$ |= $\emptyset$
			F is true for no v in t
	one v : type | formula	[exactly one element]	M[one v : t | F] = $\{\exists! M[F](e \oplus v \rightarrowtail \{x\}) \mid x \in e(t)\}$
			F is true for exactly one v in t
	lone v : type | formula	[zero or one exists]	M[lone v : t | F] = M[no v : t] $\vee$ M[one v : t]
			F is true for at most v in t
	Q disj x, y : type | $formula$	[disjoint Keyword]	M[all disj x,y: t | F]
			F holds whenever x and y are given different values drawn from e
Term	$term ::=$		
	var	[variable]	$X[v]e = e[v]$
	$term[var]$	[application]	$X[a[v]]e = \{\langle y_1, \ldots, y_n\rangle \mid \exists x.\langle x, y_1, \ldots, y_n\rangle \in e(a) \wedge \langle x\rangle \in e(v)\}$
Divers			
	$univ[expr]$	[universal relation]	contains every tuple
	$none[expr]$	[empty relation]	contains no tuple
	$iden[expr]$	[identity relation]	maps each atom to itself

Table 2.4: *Summary of Alloy Logic Syntax and Semantics. (extended from [47, 48])*

Sig t extends s {...}. Note that *sub-signatures* may themselves have sub-signatures. A set of modifiers can precede any signature declaration: (i) the *abstract* keyword that enforces that no atom is directly typed by that signature and (ii) *quantifiers operator* keyword that enforces the multiplicity (*i.e.*, the number of atoms typed by the declared signature). Without the multiplicity keyword, there can be any number of atoms typed by the signatures. The general form of the signature is as follows:

signature ::= [static] qualifier,... **Sig** sig name [type-params] [extension] decl, ... formula-seq
qualifier ::= **part** | **disj** | **exh**
type-paramaters ::= [type-paramater, ...]
extension ::= **extends** sig, ...

Facts. Is a formula used to specify constraints properties on the model elements that should always hold in instances of the Alloy module in which they are declared. It can be considered as an assumption. The fact declaration starts with the keyword *fact* followed optionally by a name and a block containing a boolean-valued Alloy expression. The fact declaration form is as follows:

fact ::= **fact** [params-name] [type-params] formula-seq

Predicates and Functions. Are both parametrisable Alloy expressions. They are declared using the *pred* and *fun* keywords, followed by an identifier, optional parameters and a block containing Alloy expressions. The predicate specifies a configurable constraint that can be instantiated in different contexts. The function is a parametrised formula that can be applied by instantiating the parameters with expressions whose types match the declared parameter types. *e.g.*, given the signature: *Sig s* and *Sig t*, a function has the form $fun\ f(arg_1 : s, arg_2 : t)\{\dots\}$. The arguments present the function inputs, and the output value represents by the whole expression application. Note that there is a convention that the *second argument* in a function declaration is treated as the function's result. There are two predefined functions and predicates: *sum* and *disj*. The *disj* predicate returns true or false depending on whether its arguments represent mutually disjoint relations, and the function *sum* takes a set of integers and returns their sum. The declaration form of the alloy function and predicate is as follows:

predDecl ::= **pred** [this arg] param name [type params] [paraDecls] block
funDecl ::= **fun** [this arg] para name [type params] [paraDecls]: expr expr
thisarg ::= sig . | sig ::
paraDecls ::= (decl,*) | [decl,*]

Assertions. Are special predicates declared with the keyword *assert*, defines a boolean formula that it is claimed to be always true. Its declaration form is as follows:

assertion ::= **assert** [param name] [type params] formula-seq

Expressions. Expressions in Alloy fall into three categories, which are determined not by the grammar but by type checking: *relational* expressions, *boolean* expressions, and *integer* expressions. The used term expression, in general, refers to a relational or integer expression. Therefore, constraints the formula refers to a boolean expression. The *logical operators* apply only to constraints, and the relational and arithmetic operators apply only to expressions. Alloy defined an exception for the two condition constructs, expressed with implies (or =>) and *else*, and the *let* syntax, which apply to all expression types.

- **conditional expression** takes the form :

  expr ::= expr (=> | **implies**) expr **else** expr

- **Let expressions.** Allows a variable to be introduced, to highlight an important subexpression or to make an expression or constraint shorter by factoring out a repeated subexpression:

  expr ::=**let** letDecl, block Or Bar
  letdecl ::= var = expr

Commands. Are a set of instructions that order Alloy Analyser to perform an analysis and generate instances for a given Alloy module. Alloy language defines two commands, *check* and *run*, which are used as follows:

- *check* command used to check an assertion; its result is a sample of counter-examples in which the assertion is violated if this is found.

  check ::= **check** [param name] [scope] [excluded] [**expect** number]
  scope ::= **for** [number **but**] typescope, . . .
  typescope ::= number (sig | **int**)
  excluded ::= **without** global, . . .
  global ::= **facts** | **constraints** | param name

- *run* command used to check a given predicate and find an instance of a function. It generates a sample of instances in which the predicate holds.

  run ::= **run** param name [scope] [excluded] [**expect** number]

Both commands take the assertion name or function and the scope indication in which the analysis is to be performed. The Alloy analysis is decidable only because it is performed on a finite domain. This is achieved by associating a scope to each module's signature, *i.e.*, an upper bound to the number of atoms typed by each signature of the module.

We note that presenting all the details of the Alloy language is beyond the scope of this work. We invite the reader to read the remarkable book of Jackson author on Alloy in [47].

2.6.2.3 Alloy Analyser

The simplicity of both relational logic and the language presented above as a whole makes Alloy suitable for automatic analysis. Alloy is supported by *Alloy Analyser*. It is a tool yielding a set of Alloy instances model whose elements are typed by concepts and relations of the meta-model defined by the Alloy module and satisfy constraints of the module. An instance is obtained by checking an assertion. It is called a counterexample if it is obtained by violated assertion checking. The Alloy Analyser can list instances or counterexamples so that it produces a different result each time.

Alloy Analyser is a bounded model checker using SATisfiability (SAT) solvers. It represents a powerful analysis tool that allows one to search for specification instances and check models' intended properties by resorting to SAT solving. It is an automatic tool that translates the alloy specification into FOL expressions as Boolean Satisfiability Problem (SAT) using the Kodkod model-finder[50]. Then, this problem is solved by an SAT solver, *e.g.*, SAT4J [51], MiniSat [52], or Berkmin [53], and the result of the analysis is displayed to the user. Boolean satisfiability is the problem of finding an assignment to a boolean formula on which the whole formula is evaluated to be true. The primary analysis technique associated with Alloy is essentially a counterexample extraction mechanism based on SAT solving. Given a system specification and a statement about it, a counterexample of that statement (under the assumptions of the system description) is exhaustively researched.

Thus, the *Alloy Analyser* is a constraint solver that provides two types of automatic analysis the simulation and verification. (i) The *simulation* analysis is supported using the keyword *run* to produce instances of the model (Alloy specification) satisfying a condition. (ii) The *verification* is based on a model-finding approach using an SAT solver. To support the verification, alloy Analyser uses the keyword *check* to check whether an assertion holds for a specific scope of atoms. An assertion differs from a fact in that the *Alloy Analyser* will check an assertion to see if it is true for all the examples in the scope. In contrast, the *Alloy Analyser* assumes each fact is true and uses them to constrain the examples it examines. If it finds a counterexample, then the predicate is *unsatisfied.* If no counterexample is found, the predicate may be either *valid* (*i.e.*, true for all possible examples), or it may be *unsatisfied* but not within the used scope.

As FOL is not decidable (and the relational logic is a proper extension of first-order logic), SAT solving cannot generally be used to guarantee consistency (or, equivalently, the absence of counterexamples for) of a theory. Then, an exhaustive search for counterexamples has to be performed up to a bound k in the model elements (signatures) to limit the domain of the interpretations. The bound is a positive integer restricting the number of instances of each instance element of the analysed system. Thus, if no counterexample is found, the assertion could still be invalid for an upper bound. In the same way, in the case of a simulation (run), if no instance is not found, the condition could be valid for an upper bound. Therefore, this analysis procedure can be regarded as a validation mechanism rather than a verification procedure. Its usefulness for validation is justified by the exciting idea that, in practice, if a statement is not valid, there often exists a counterexample of it of small size.

Finally, It is important to note that the authors of Alloy provide the Java API on which *Alloy Analyser* was built. As a result, it is straightforward to integrate the analytical power of Alloy in tools.

2.7 Summary

We have presented throughout this chapter the basic notions of BPM and the main standard notation for modelling business systems BPMN. Secondly, we have presented an overview of the existing verification methods. Thirdly, we present the FOL logic, which is used in this thesis as a basic language to specify the execution semantics of the BPMN. Finally, we have presented TLA$^+$ and Alloy, which are two formal specification languages based on the FOL logic, are increasingly popular due to their simplicity and flexibility. They constitute a powerful but simple language associated with effective and automated tools. Table 2.5 summaries their particularities. In the next chapter, we describe an exhaustive overview of the verification works for the BPMN standard.

	Alloy	TLA$^+$
Modeling	Relation logic	First-order logic Actions + Fairness
Specification	Relation logic	Temporal Logic
Verification	Bounded model checker (Analyser)	Unbounded model checker (TLC) (+ Theorem prover (TLAPS))

Table 2.5: *General Comparison of the Alloy and TLA$^+$ Tools. (from source table [54])*

Chapter

3 Literature Review

“*Education is what remains after one has forgotten what one has learned in school.*”

Albert Einstein

Chapter content

3.1 Introduction

Business process verification is a valuable phase to avoid defaults at design time rather than when running the processes over business process engines. BPMN is the most prominent notation for representing business processes due to its wide usage in academic and industrial contexts. However, since BPMN suffers from using the semi-formal definition of its execution semantics, its formalisation was a research question. This chapter aims to provide an entirely comprehensive and detailed high-level survey of the significant works on formalising and verifying BPMN models. It gives students and researchers in this domain a helpful base for their future research and guides us to evaluate our proposal regarding existing work. We conducted a systematic literature review to collect and analyse the work on business process formalisation and verification present in the literature and analysing supporting tools. We investigated 1445 papers, and we identified a count of 79 Business process verification and tools works for two kinds of business process diagrams. The authors report direct or indirect formalisation and/or its implementation using different formal languages for each of these works. Compared to prior systematic literature on business process verification and tools, this chapter extends the current knowledge [55–58] by analysing works based on different formalism and focused on various business process perspectives.

This chapter is organised as follows. Section 3.2 describes the protocol of our systematic review, including research objective, research questions, research strategy, how we extracted and classified our data (inclusion, exclusion criteria and quality assessment criteria). Section 3.3 describes the selected approaches for the verification of the BPMN models. Section 3.4 presents our analysis, comparison and discussion of the studies and, finally, a conclusion is given in Section 3.5.

3.2 Research Method

This section describes the review protocol we adopted for the literature search for selecting the relevant articles. We chose the approach of Petersen *et al.* [59] to conduct our systematic review. In turn, we discuss the underlying research questions, preliminary research, search strategy, inclusion, exclusion and quality assessment criteria.

Figure 3.1 summarise the approach application. First, we define a set of research questions. Second, depending on the questions that we asked, we choose terms to build our search queries. Third, we use these queries in the search engine of selected databases. Fourth, after collecting the results of each database, eliminating redundant papers (*i.e.*, journal or conference papers indexed into more than one database). Then apply the first screening phase in which we read for each article only the title, the summary, and the keywords, and we see if they can be included. Afterwards, we pass the selected papers for a second screening phase where we apply a set of inclusion, exclusion, and assessment quality criteria that we define. Finally, the output papers will pass for the analysis and the extraction of data. Figure 3.1 gives an overview of the number of papers after each step of the article selection.

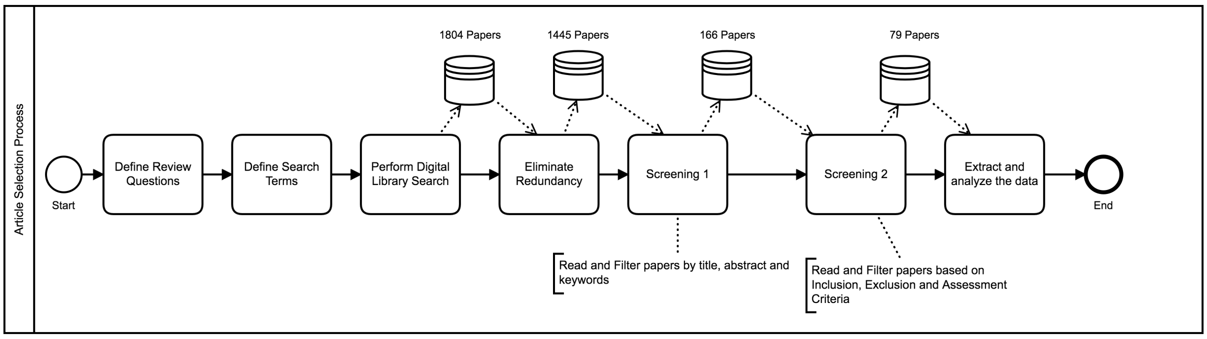

Article Selection Process
1804 Papers
1445 Papers
166 Papers
79 Papers
Start
Define Review Questions
Define Search Terms
Perform Digital Library Search
Eliminate Redundancy
Screening 1
Screening 2
Extract and analyze the data
End
Read and Filter papers by title, abstract and keywords
Read and Filter papers based on Inclusion, Exclusion and Assessment Criteria

Figure 3.1: *Article Selection Process.*

3.2.1 Objectives

This chapter aims to accumulate an understanding of the current state-of-the-art in the verification of the business process models domain, summarise existing techniques and tools, and identify the areas of further work.

3.2.2 Survey questions

Describing the underlying research questions is the first step of any systematic literature review to guide the identification and analysis of studies. To achieve the objectives mentioned above, we have formulated the following questions:

- Which formal model languages are used to formalise the semantics of BPMN ?
- What are the goal(s) of this formalisation ?
- Which is the state of tool support ?
- What are the parts of BPMN being supported by the tools ?
- Which are the challenges that still need to be addressed ?

3.2.3 Prior Reviews on Business Process Modelling Verification

We are not the first ones to conduct a systematic review on the verification of the process models. So far, there have been several reviews about topics related to process model analysis in general (*e.g.*, [55, 57, 60–62]). Nevertheless, none of these reviews studies the works based on the formalisation of BPMN execution semantics models and their verification tool support.

In [60], the authors contributed in the area of *Event-driven Process Chain (EPC)* verification. They analyse 712 models with three tools (EPCTools, ProM plugin for EPC soundness analysis, and YAWL Editor) for checking soundness properties. In [61], the authors investigated EPC approaches and tools for addressing the problem of automating business processes. They study the modelling and verification of business requirements in the domain of EPC and discuss the verification methods used in the design of IoT systems. Unlike this work, we based on BPMN as the modelling language.

The work that comes closest to the ambition of our review is [55], which provides a survey of formal verification approaches for business process models. The selected work was published before 2008 and was generally focused on BPMN or activity diagrams of UML as notations for business processes modelling. Unlike our survey, we focus only on BPMN as a modelling language and the work from 2008. For that, we can consider that this survey is complementary to their work.

The work in [62] presents an analysis of verification tools introduced in different application areas: variability (*i.e.*, whether works based on a business process with a characteristic of supporting various versions depending on the intended use or execution context); compliance (*i.e.*, whether process models conform with specifications), and compatibility (*i.e.*, whether two process models conform semantically). Unlike our work, the selected work was published in [2005-2008], and none of them verified the order or parallel execution of Message Flows, Tasks, Events, etc., in a Collaboration. In addition, the used modelling language in the selected set of work is not restricted to BPMN.

Recently, some systematic reviews have been presented in [57], and [63] where they focus on BP compliance. More precisely, the authors in [57] conducted a state-of-the-art of business process compliance approaches published in [2003-2013]. They classified the compliance over four dimensions (scope, lifecycle phase, formality and contribution type of compliance approaches) where the formalisation of the business process represents only a sub-dimension of the formality dimension. The authors in [63] presented a systematic review focusing on the management of business process compliance requirements (data flow, control flow, resource, allocation, time). They addressed works published in [2011- 2017]. The authors did not focus on BPMN as a modelling language.

Our literature review builds on the one in [55] including the references reported in their review that were related to process modelling verification based on BPMN. Beyond that, we took into account newer works from the year 2008 to January of 2021. We have also defined our own more specific search string and research classifications for works.

3.2.4 Research Strategy

Given the broad nature of the research domain, finding all relevant papers by manually searching through conferences and journals would have been very time-consuming. Therefore, we opt to start the search process with an automated search. The literature collection process started by querying prominent scholarly databases. For this purpose, we first identified a list of subject terms, concepts and keywords. Next, we determined six primary keywords from our review questions: "BPMN", "Verification", "Formal semantics", "Framework", and "Approach". We then added other keywords that are synonyms or related to those five:

- an alternative terms for *"BPMN"* are *"BPM", "Business process modelling", "Process Diagram"*, and *"Collaboration Diagram"*.
- an alternative term for *"Verification"* is *"Analysis"*.
- an alternative terms for *"Formal semantics"* are *"Formalising"*, *"Formalisation"*, and *"Formal Methods"*.
- an alternative term for *"Framework"* is *"Tool"*.
- an alternative term for *"Approach"* are *"Method"* and *"Technique"*.

These terms were then combined using boolean operators to construct the key search terms such as:

- *"BPMN" AND ("Formal semantics" AND "Approach") AND "Verification".*
- *"Business process modelling" AND ("Verification" AND "Framework").*
- *"Business process modelling" AND ("Verification" OR "Analysis") AND ("Framework" OR "Tool").*

- *"Verification" AND ("Technique" OR "Method" OR "Approach")* AND *"Collaboration Diagram"*.

As a final check, the search string was updated and re-run to reflect frequently occurring words in the titles, abstracts, keywords of relevant papers found through reference search results. Consequently, we determined that "BPMN semantics" and its related keywords must be included in the papers' title or abstract. Our focus is on all articles published in English between 2008 and January 2021. Therefore, the metadata fields we used are title, abstract and keywords. These yielded the following:

- *Search string= ("Business Process Modelling" OR "BPMN" OR "BPM") AND ("Verification" OR "Analysis") AND ("Formal Semantics" OR "Formalising" OR "Formalisation" OR "Formal Methods") AND ("Technique" OR "Method" OR "Approach") AND ("Framework" OR "Tool")*
- *Title= "BPM" OR "BPMN" OR " Process Model" OR "Process Modelling "OR " Process Modelling" OR "Process Diagram" OR "Collaboration Diagram"*
- *Document Types = MEETING OR CONFERENCE OR ARTICLE*
- *Timespan = 2008-2021*
- *Search language = English*

To perform an exhaustive search, we have identified five electronic sources of relevance where major conferences and journals in the domain publish their proceedings: SpringerLink[1], IEEExplore[2], ACM[3], WEB of Science[4], ScienceDirect[5], and ScienceScholar[6] databases. Table 3.1 presents the number of search results per database. The literature search resulted in more than 1445 papers. The first column presents the electronic databases; the second column shows the number of articles found on each given database after searching and eliminating the redundancy papers. The third column presents the number of papers that are kept distributed against each database after the first screening phase. The fourth column presents the final number of papers preserved after the second screening phase distributed against each database. Then, the following columns present the distribution of these papers in terms of their types of Conferences, Workshops, Symposiums or Journals. The last column presents the percentage of included papers per database.

Table 3.1: *Number of Studies per Database.*

Electronic Data Base	Number of retrieved Papers	Number of initial selected papers	Number of final included papers	Conferences	Workshops	Symposium	Journals	Percentage in final inclusion(%)
IEEExplore	40	28	23	13	2	2	6	29.11%
ACM Digital library	8	4	2	0	1	0	1	2.53%
ScienceDirect	464	15	11	0	0	0	11	13.92%
SpringerLink	872	84	21	13	5	1	2	26.58%
WebOfScience	77	41	19	6	3	0	10	24,05%
ScienceScholar	24	5	3	0	1	0	2	3,79
Total	1445	177	79	32	12	3	32	100%

3.2.5 Article Selection and Inclusion and Exclusion Criteria

The collected papers were then checked for their relevance to the literature review by checking their titles and reading their abstracts to ensure that they are related to business process modelling and verification of BPMN topics. If the paper passed these preliminary checks, it was included in the pool. Those papers whose title was not relevant were immediately removed from the analysis. Initially, we identified 177 papers as potentially applicable to the research questions. Then, we manually analysed the selected articles. To obtain papers directly contributing to our research questions, we further processed the elimination of papers by considering the inclusion and exclusion criteria. Each paper satisfying at least one inclusion criterion and not meeting any exclusion criteria and satisfying all the quality assessment criteria was selected to be read in its entirety in the next step. More specifically, we removed and included papers as follows:

[1]SpringerLink : https://link.springer.com
[2]IEEExplore: http://ieeexplore.ieee.org
[3]ACM Digital library:https://dl.acm.org/
[4]WebofScience: https://apps.webofknowledge.com
[5]ScienceDirect: www.sciencedirect.com
[6]Science Scholar:https://sciencescholar.us/

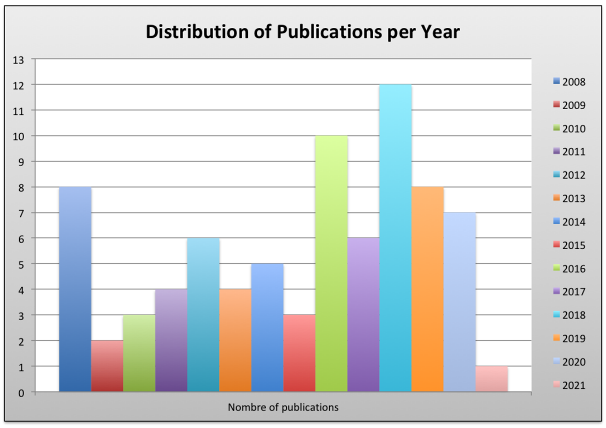


Figure 3.2: *Distribution of Selected Papers per Published Year (2008-2021).*

- **Inclusion Criteria**
  - Papers published between 2008-2021.
  - Papers able to answer the research questions.
  - Papers that fall in business process model execution.
  - Papers that fall in transformations from one language to another.
  - Algorithms for the formal verification of process models for *e.g.*, deadlock or safety.
  - Papers deal with the syntax and syntactical correctness of process models.
  - Papers deal with the semantics and semantic correctness of process models
  - Papers that are published in English language.
- **Exclusion criteria**
  - The publication was published in a language other than English.
  - The full text of the publications was not available.
  - The publication did not coincide with the topic of systematic research.
  - BPMN was used only as a presentation tool and not as part of the research.
  - Any duplicate papers, industrial bulletins, industry case studies.
- **Quality Assessment criteria**
  - The study objectives are clearly stated.
  - The proposed method / technique is clearly described.
  - The methodology used in the study is adequate.
  - The study has a high citation.

After applying the inclusion, exclusion, and quality assessment criteria, only 79 papers remained in the set of relevant papers. The number of false positives in the initial set (papers that may have been relevant but on detailed investigation turned out not to be so) was disappointingly high. Figure 3.2 shows the distribution of selected papers per year; we can see that the subject of formalisation and verification of business process models represents a high topic in the period coinciding with the present thesis.

3.3 Papers Overview

Several works have investigated formal or semi-formal approaches to provide a method for verifying and validating BPMN business process models. This section serves as a survey for the BPMN verification methodologies in the literature. We will classify these works into five categories: approaches based on *Petri Nets, Automata Theory, Process Algebras, Logic Formulas, and a Programming Language.* In the following subsections, we give an overview of each work in each category. To help readers, we present in Table 3.2 and Table 3.3 an abstract definition for the languages and the technologies used among the selected works, respectively.

3.3.1 Approaches based on Petri Nets

Petri nets are used to formalise and verify the correctness and soundness properties of BPMN in several works. Among them, one of the earliest works that we identified is [64]. The authors propose a transformation from BPMN 1.2 into Petri nets models. They have offered a set of rules which specify how to transform one or a combination of BPMN element(s) into a Petri-net module. Firstly, they have chosen the *ILog BPMN Modeller* as a graphical editor to create BPMN models. Then, they have provided the *Transformer* [65] tool where they apply the *BPMN2PNML* transformation rules on BPMN models and export the resulting Petri net in the form of a *PNML* file. Then, they use the resulting files as input for the verification tool *ProM*[66] to statically check the semantics of BPMN models. With the ProM framework, soundness and other properties such as dead transitions, deadlocks, and livelocks are verified. However, even if the work deals with collaboration elements, the formalisation as Petri nets suffers from limitations for the communication presentation, the hierarchical relation between processes and subprocesses, and the verification of internal activities within them. In [67], the authors then build on their earliest work and propose formal semantics for the transaction and compensation elements of BPMN in terms of Petri nets.

In [68], the authors propose an approach for the verification of business process models based on the transformation technique. They convert a BPMN model to an extended *Business Process Execution Language for Web Services (BPEL4WS)* model manually. Then, the *BPEL4WS* translated to a Petri-net model in a *Coloured Petri net (CPN)* XML-based representation. Then they use *CPN* tools to verify the model properties (deadlock and infinite loop). The proposed approach is semi-automatic since the first transformation is manual.

In [69], the authors present an approach for automated model-checking and analysis of entailment constraints (i.e., the dependency between tasks) in the context of business processes to detect deadlocks and security property violations. First, the authors annotate the BPMN business process model with security artefacts. Then, they describe formal semantics in terms of *CPN*. For verifying the security properties, they define a translation of a BPMN process model into the *PROcess MEta LAnguage (PROMELA)* in which a *CPN* semantics is implemented. Next, security properties are verified as a set of linear temporal logic formulae using the *SPIN* model checker. Finally, the authors developpe a model checking plug-in for the free web-based process modelling tool, called *Oryx*. This tool allows the automated translation of security annotated process models into *PROMELA*, the simulation and verification of security properties defined in a simple dialogue box, and the reporting of verification results back to the modeller.

In [70], the authors provide a formal semantics for a subset of BPMN in terms of graph rewriting rules using an algebraic graph transformation approach to describe direct traceability between the statics structure. Then, in [71] the authors extend their work and propose a formalisation of a larger set of BPMN 2.0 based on in-place graph transformation rules. Then, they have documented it visually using BPMN syntax through *Graph Rewrite Generator (GrGen.Net)* platform, which helps to debug and simulate the execution of the business process models. The authors do not treat verification aspects, but they clearly map mathematical sets and relations to the specific graph-oriented concepts from their reference implementation.

In [72], the authors propose an approach for detecting errors in business process models using the *influx* tool. First, they use *influx* for modelling process requirements and generating *XMI* files. Influx models are based on BPMN 1.0. Then, the authors introduce a method for checking both syntactic and structural properties of the generated model. To check syntactical errors and the presence of cycles in a model, they use a simulation. To check structural errors, they segregate models into Workflow-Net and Petri nets. They then use either *Woflan* [73], or *Lola* [74] to check for soundness (*e.g.*, for the presence of deadlock and lack of synchronisation). Finally, the authors extend their work in an empirical study in [75] to find out syntactic and control flow error frequencies in industrial process models using graph-theoretic techniques and Petri net-based analyses. Also, they have studied the connection between errors (syntactic and control flow-related) and a set of metrics related to structural and behavioural aspects of process models. To detect control flow-related errors, they use *Lola* and counterexamples provided by this tool. They show that up to 92.9 % of process models are erroneous in business contexts.

In [76], the authors propose a framework based on formal verification for detecting and diagnosing errors in business processes. Firstly, the authors use graph search algorithms to detect syntactic errors, remove them to get a well-formed model. Then, they apply a transformation from the well-formed model to *PNs* preserving soundness properties. Then, they reduce the *PNs* to *Workflow-net* for checking its soundness using the *Woflan* tool. Then, based on the diagnostic information given by the tool, they identify the location of errors. In [77], the authors extend their work with rigorous proof of the results and an extensive empirical analysis of diagnosis of errors. They analyse a sample of 174 industrial BPMN models in which they identify more than 2000 errors. The authors focus on the statistical model of error probabilities at subprocesses.

In [78], the authors propose an approach to enable checking data-flow correctness for process models. The proposed work is limited to some essential control flow elements and data objects in processes without any data values. First, the authors provide an algorithm for mapping the process models in an extension of *Workflow Graphs* enhanced with data. Then, they provide an implementation of the mapping algorithm for BPMN, and OTX [79] [7] models. Then, they provide a second mapping from the *Workflow Graphs*[8] to *PNs* to support data-flow verification in a process. Next, they define a set of data-flow correctness properties (missing data, redundant data, lost data, and inconsistent data) as *anti-patterns*[9]. Finally, they check the correctness of these properties using *Lola* model checker.

In [80], the authors propose a data-value-aware verification approach. They introduce a transformation algorithm from data-value-aware BPMN process models to *PNs*. The BPMN models are enhanced with information on data values using so-called Data-Value Enhancement Functions. These later facilitate specifying the usage of data values and their modifications during the process flow. To verify process models, they rely on the model checking of Petri Nets using the *LOLA* tool. In their work, they focus on data-value centred properties specified in *Computation Tree Logic (CTL)*. Their approach is evaluated on an auction model. This approach is limited to tiny models due to state-space explosion.

Recently, the work in [81] has been extended the approach in [64] to *Timed Petri Nets (TPNs)* to be able to express time constraints for activities, process regions, and timer events. Besides, they tackle the detection and management of constraint violations at run-time. The authors propose this formal model to analyse the soundness of the collaboration and its temporal constraints. However, no formal verification means for these properties is given.

In [82], the authors define a formal semantics of a subset of BPMN that covers timer events, tasks, and exclusive and parallel gateways. They propose a transformation approach from the BPMN models to TPNs to detect control flow anomalies in the model. Then, the authors check liveness and reachability properties regarding the execution time by using the *Timed Petri Net Analyser (TINA)* tool. The authors focus on the timer events (boundary, intermediate) with an associated duration time. According to the authors, to deal with different perspectives of BPMN, like data and communication, they would have to move to another extension of Petri Nets like *Interval Timed Coloured Petri Nets (ITCPN)*.

In [83, 84], the authors propose an approach for transforming BPMN models limited to a small subset of control flow element to *CPN* models using *GRaphs for Object-Oriented VErification (GROOVE)* tool. They provide a new method for verifying the preservation of the semantics and its validation by ensuring the absence of the live-lock, the improper structure and loop, and the deadlock. The authors have verified the successful termination of the transformation using the *GROOVE* model checker. As far as verification properties are concerned, they focus on safety and vivacity using *CPN* Tools.

In [85, 86], the authors propose a formal semantics for a subset of BPMN using high-level Petri nets, called *RECursive Petri nets (RECATNets)* and *Rewrite Logic (RL)*. The authors provide a prototype to perform the automatic transformation from BPMN to *RECATNets* using a specific *ATLAS* transformation language. Then, the obtained *RECATNets* are translated into *RL* terms, and the Maude model-checker is used to verify proper termination and some other Linear Temporal Logic (LTL) properties. The work covers the behaviour of the subsets of BPMN elements, including multiple instantiation, cancellation of subprocesses, and exceptions. No information about the communication model is given, and only a small set of BPMN elements is covered. In addition, no time semantics is given for the timer elements mentioned. Besides, the approach is semi-automatic; neither the implementation nor benchmarking are given, and the approach is illustrated only through three simple examples.

In [87], the authors propose a transformation approach for formalising a subset of BPMN elements extended with time and probabilities. The authors apply two mapping approaches, one automatic for the simulation and another manual for the verification. The authors develop a package for the mapping of BPMN models into Petri nets automatically. This transformation extends the BPMN models by

[7]OTX: is an industry-standard for modelling commissioning processes of vehicles.

[8]Workflow graphs: represent the primary control-flow constructs of industrial process modelling languages such as BPMN, EPC and UML activity diagrams.

[9]Anti-patterns: are specific patterns in software development that are considered bad programming practices.

additional probabilities on the gateways, time delays on tasks. Then, they analyse the resulted Petri net XML file in the *General purpose Petri net simulator (GPenSIM)* [88] tool. As far as verification purpose, which *GPenSIM* lacks, the authors apply a manual transformation from the BPMN models to *PNs* according to a set of defined mapping figures. Then, they analyse the transformed Petri net and verify generic properties of Petri nets (liveness, safeness, ordinary, pure, invariants, free choices, boundness) using *TINA* tool. In this paper, the authors use the timed Petri nets Analyser for verifying the extended Petri nets without its definition. In addition, they use for the automatic approach the *GPenSIM* tool, which is not a model checker. In addition, they use the *TINA* tool for the verification without any precision about using the Bounded Prioritised Model-checking.

In [89], the authors present a transformation approach of BPMN models into *CPN*. They have developed a tool called *Coloured Petri Net for BPMN design (CPN4BPMN)* to automatise the transformation. Their approach operates in two phases. Before applying the transformation, a BPMN design tool (such as the BPMN 2.0 designer of the Eclipse IDE) is used to partition a BPMN model into sub-models. Then, they use the sub-models as input for the *CPN4BPMN* tool to generate the corresponding *CPN* model. Even if their work covers a large set of BPMN elements, message exchanges are not specified. The authors extend their rules in [90] to handle both structured and unstructured BPMN models.

In [91], the authors propose an approach for assisting business process designers in identifying necessary cloud resources concerning temporal and financial restrictions on the process. The authors extend BPMN models by enriching process activities with temporal constraints and required cloud resources. To perform the verification of such models, they propose an automatic generation and conversion of the enriched BPs into *TPNs* using *Extensible Stylesheet Language Transformations (XSLT)* as a transformation language. Afterwards, they check the BP using the *TINA* model checker. This approach is evaluated on a simple case study.

In [92], the authors propose a new approach for verifying the *k-soundness*[10] of BPMN models taking into account cross-case data objects (i.e., data objects shared among different process instances of more than one participant). The authors based on existing works for the transformation of BPMN into *CPN* and introduce a set of rules for mapping BPMN elements including data objects to *Hierarchical Coloured Petri nets (HCPN)*. Then, based on the mapping, different correlation mechanisms are defined that relate data objects to cases [11]. The BPMN standard supports two different correlation types: key-based and context-based correlation mechanisms. The authors discuss several correlation mechanisms by considering combinations of basic ones: singleton/any correlation, key/context-based correlation and case-based correlation. For the verification of the *k-soundness* property, the authors implement a search function in *CPN* Tools. They provide a compiler, called *fcm2cpn* [93], for automatic translating process fragments to CPN tools compatible to Petri nets.

Other transformations that have been proposed are those from BPMN to workflow models, *e.g.*, *Yet Another Workflow Language (YAWL)* [94–96]. The authors provide a formal semantics of BPMN models in terms of mapping to YAWL nets. This mapping is given as an algorithm based on a set of syntactic rules. Then, they implement their proposal as a plugin, called *BPMN2YAWL*, integrated into the ProM platform. This plugin transforms a BPMN model into a *YAWL* as an *XML* file. This *XML* file then serves as input to a *YAWL-based* verification tool. As a proof of concept, they test the tool using simple models. As far as verification is concerned, the authors focus on deadlock freeness and soundness properties.

3.3.2 Approaches based on Automata Theory

Other approaches rely on automata theory to define the semantics of BPMN formally.

In [97, 98], the authors propose an approach to detect structural errors in business process models based on model checking. First, the authors translate the model in *PROMELA*. Then, they map it to a *Kripke Structure (KS)* to express the behaviour of the process models using the *SPIN* model checker. Once the KS is obtained, the authors check properties (absence of deadlocks, lovelock and multiple terminations) expressed in LTL formulas and determine a set of structural errors patterns. The authors use a non-deterministic automaton (as an external technique) representing the aimed model behaviour to identify structural errors. Finally, they implement their translation rules in a java plugin, called *ESPIN*. In their work, the authors focus on a small subset of BPMN elements (*i.e.*, parallel, exclusive gateways, abstract tasks and none start/end events).

[10]k-soundness: no deadlocks, no dead transitions, and no remaining tokens property, where k is the number of process instances.

[11]Cases: different process instances

In [99], the authors propose an approach for verifying business process models enriched with temporal and resources constraints. First, they provide an automatic mapping from BPMN extended with static time constraints and resource annotations onto *Timed Automata (TA)* networks. Then, they use the *UPPAAL* model checker for the verification of the generated models w.r.t. deadlock and bottleneck properties.

In [100], the authors propose an approach for the performance evaluation of business process models. First, the authors impose structural restrictions to assume that the models do not contain unreachable activities and can permanently terminate. Then, they propose an algorithm that allows for a transformation from restricted BPMN models to *Stochastic Automata Networks (SAN)*. This algorithm is implemented as part of a tool called *BP2SAN*. Finally, for the performance analysis, the authors evaluate how the time of the process and the resources utilisation may be impacted the system workload using the generating *SAN* models in *Performance Evaluation of Parallel System (PEPS)* tool.

In [101, 102], the author proposes a transformation from BPMN models to *TA* networks. Where the behaviour of each task in the BPMN model is mapped to a *TA* network. The author supports communication by modelling the collaboration model as a network of several synchronised TA networks. Moreover, he requires timer events in models to be used to determine the time constraints in them. The author uses the *UPPAAL* tool to simulate and verify the generated TA networks. Properties of interest are expressed in *Clocked Computation Tree Logic (CCTL)* formulas. To support the transformation from BPMN to *TA*, the author uses the latest version of the *BPMN2TA* plugin in the *BTransformer* tool [103]. In these papers, the author uses the synchronous communication model, which is provided by the semantics of the *TA* network. However, communication in BPMN reflects the real-world collaboration between distinct business processes where this may not be respected. Also, the author uses time durations for all BPMN timer elements, which do not respect the semantics of timer BPMN elements and their types provided by the standard. Further, No transformation rules are provided in the paper.

In [104], the authors propose a formal specification and verification approach of advanced temporal constraints for business processes, based on *TA* networks. The authors provide a specification for relative and absolute related temporal constraints while relying on the dependencies between these constraints. Before transforming BPMN models, the authors extend them with annotations to specify time constraints. The definition of temporal constraints allows one to specify constrained process models that may encounter a deadlock situation due to inconsistencies between nested temporal constraints. Then, the authors apply a set of transformation rules given as templates between the BPMN elements and *TA*. Finally, they use *UPPAAL* model checker to perform the formal verification of the timed business processes. In this work, the authors extend the BPMN time semantics by associating time intervals to tasks, to events, and in between activities. The authors support time cycles by using an equivalent model of a loop task associated with a boundary timer event.

In [105], the authors propose an approach for the verification of BPMN models w.r.t. formalised business rules. They present a mapping from BPMN elements to asynchronous *Finite State Machines (FSMs)* synchronised by signals. They use the *New Symbolic Model Verifier (NuSMV)* model checker to verify the specified business rule properties. However, the approach covers only a tiny subset of BPMN, including the control flow elements. Moreover, no communication semantics is provided. Indeed, complex processes cannot be translated like hierarchical business processes.

In [106], the authors propose a checking technique for the analysis of interoperability properties of collaborative processes. They present a transformation from BPMN models to *TA*. A repository of interoperability requirements is provided and used to detect problems in business process collaboration models. The authors focus on the application of the *UPPAAL* model checker to verify interoperability requirements for a given collaborative BPMN model. The formulation of interoperability requirements is given in *Timed Computation Tree Logic (TCTL)*. In this work, the authors provide users with a set of template models to facilitate the behavioural description of their process model. To provide collaboration semantics, the authors use synchronous communication between business process templates. They propose extended semantics for the tasks, communication elements, timer events, and inclusive gateway. For the tasks, they suggest the use of resources synchronisation. The sending and the reception of messages then rely on a resource usage pattern. As far as timer events are concerned, the authors support the start and the intermediate timer events based on the clock semantics given by *TA*, and the timer elements are transformed to clocks associated with the start behaviour of the tasks. For inclusive gateways, the authors simulate their local semantics based on a transformation of the model to an equivalent one based on exclusive and parallel gateways.

In [107–109], the authors propose a verification framework for business processes models, called *VBPMN*. The authors define a semantic with a BPMN to *Labeled Transition Systems (LTSs)* transformation, obtained by sequencing two successive transformations. First, a transformation from BPMN to *Process Intermediate Format (PIF)* which is next transformed to *LOTOS New Technology (LNT)*, (LNT having an LTS semantics), to be fed to the *Construction* and *Analysis of Distributed Processes (CADP)* verification tool. A small subset of BPMN control flow elements is selected. It includes neither communication nor temporal elements. Also, the authors use multiple transformations, which make it unsure whether BPMN semantics is respected.

In [110], the authors propose a methodology for the specification and the verification of business processes based on the use of BPMN and a refinement approach to reduce the complexity of modelling workflow systems and facilitate their understanding. First, they introduce four refinement patterns that specify alternative semantics for complex models: sequence pattern, exclusive pattern, parallel pattern, and iterative pattern. Then, they give formal semantics for the BPMN elements based on these refinement patterns and *KS* models. For verification purposes, the authors convert the *KS* models to *NuSMV* code. Then, they use the *NuSMV model checker* to verify the refinement correctness properties specified with *LTL*, such as refinement safety.

In [111], the authors propose an automatic verification tool for BPMN models. They define an executor and verifier tool based on the OBP technology [112]. Their approach is based on three steps. Firstly they define a BPMN extractor to generate *Property Sequence Chart(PSC)* models from BPMN models. Then, they give a semantics for the generated PSC by transforming them to *Büchi automaton (BA)*. Finally, to perform the verification, the authors express properties in the *Generic Property Specification Language (GPSL)* language [113]. Unfortunately, the authors do not give any details about the transformation approach in terms of formal semantics.

3.3.3 Approaches based on Process Algebras

Likewise, tool-supported methods that rely on process algebra as a semantic formalism for BPMN models are advocated.

In [114], the authors present a formal semantics for BPMN with an encoding into the *Communicating Sequential Processes (CSP)* process algebra. They also show how this semantic model may be used to verify that one BPMN process diagram is consistent with another one. The authors extend their work in [115] to verify compatibility between business participants in a collaboration using the *Failure Divergence Refinement (FDR)* tool. Nevertheless, the work does not provide an environment for verification. In [116] the authors again extend their work to propose a relative timed semantics for BPMN models. They augment their model by introducing the notion of relative time in the form of delays chosen non-deterministically from a range. The authors adopt a variant of the two-phase functioning approach widely used in real-time systems and coordination languages like Linda. In [117], the authors document the relationship between compatibility in the untimed and timed settings for BPMN models and present a pattern-based approach to construct BPMN property specifications.

In [118], the authors propose a direct translation of BPMN models into the *Calculus of Orchestration of Web Services (COWS)* extended with probabilities. This approach enables the derivation of a COWS specification from XML representations of BPMN models provided by modelling applications. The authors focus in their work on quantitative properties based on probabilities on the extended model, using the *PRISM* model checker.

In [119], the authors propose an automated transformation from an extended BPMN to *Timed CSP (CSP+T)*, as well as composition verification techniques for checking properties using the *Failures Divergence Refinement (FDR2)* model checker tool. They focus on the semantics proposed in [116]. As far as the verification properties are concerned, the authors rely on liveness and reachability.

In [120, 121], the authors present a framework, called *Stochastic BPMN Analysis Tool (SBAT)* for the modelling and analysis of complex business workflows. The authors extend the BPMN models with probabilistic and non-deterministic branching and reward annotations for enabling the translation into a series of *Markov Decision Processes (MDP)*. Then, the generated models are used to check formulas expressed in the temporal logic formalism, *Probabilistic Control Tree Logic (PCTL)*, using the stochastic model checker *PRISM*. The proposed framework focuses on quantitative safety requirements (timing, occurrence and order of events, reward values, transient and steady-state probabilities). It does not seem to be able to deal with qualitative requirements. In [122], the authors extend their work and present an industrial framework, called *Stochastic BPMN Optimisation and Analysis Tool (SBOAT)*, for the automated restructuring of stochastic workflows to reduce the impact of faults. *SBOAT* tool uses a stochastic model checking for evaluating the behaviour of the workflow. This latter allows for the computation of exact values of real-valued quantities modelling resources associated with the workflow.

In [123], the authors propose a model-based *Discrete Event System Specification (DEVS)* formalism for modelling, analysing, and checking temporal constraints in business processes at the earlier phase of design. Their approach handles a rich set of temporal properties: Intra-activity, Inter-activity, and Inter-processes time constraints. First, the authors take a BPMN model and a set of transformation rules as inputs. The transformation associated to each element where time constraints are used a control model, they obtain a set of atomic DEVS models. Then, they check the soundness of the resulted model using *DEVS-Suite* simulator, which is based on *JAVADEVS* models. Their simulation takes into account the generated control model for the authorisation of the execution. Based on the simulation outputs, they observe the behaviour of their proposed model relating to the temporal constraints and make conclusions about the properties checking. The authors rely on extended semantics for the BPMN elements to present the temporal constraints in their work. In addition, no explicit formal semantics is given. Last, their results are based on a simulation approach rather than a verification.

3.3.4 Approaches based on Logic Formulas

Approaches that exploit symbolic encoding as a semantic formalism for BPMN models based on symbolic verification techniques are also advocated.

In [124], the authors propose a semantic model for BPMN 1.0. The paper introduces a semantics-preserving method for transforming a subset of BPMN notational elements into the NuSMV input language based on a set of formally defined translation rules to verify the models. *NuSMV* is based on symbolic model checking. For that, the correctness of the specification of the BPMN model is expressed in NuSMV using several properties expressed as CTL formulas, and that is verified automatically using the model checker.

In [125], the authors give an execution semantics of BPMN elements expressed using *LTL*. The formalisation is defined for a large set of BPMN 1.2 elements. They evaluate their approach on a use case with deadlock and liveness analysis. The authors use message /timer intermediate events; however, no communication or temporal semantics are given. The proposed approach gives an unambiguous definition of the execution semantics of BPMN diagrams. It could serve as a basis for the formal analysis of BPMN diagrams, including control flow elements, but no tool is presented.

In [126], a BPMN formalisation in the context of conformance verification between global and local process models is provided. The authors propose a mapping from BPMN Collaboration diagrams and processes to *LTL* formulae. They follow the work supplied in [127] where BPMN workflow specifications are considered as possible visual alternatives for *LTL* formulae and an *LTL* semantics for BPMN 2.0 is provided. Then for the conformance verification, they use the *GROOVE* graph transformation tool. The presented *LTL* formulae seem only capable of capturing liveness requirements but not safety and soundness ones.

In [128], the authors provide a formal specification of well-formed BPMN processes in rewriting logic using *Maude*. They offer new mechanisms to avoid structural issues in workflows such as flow divergence by introducing the notion of a well-formed BPMN process. They focus on data objects semantics and their use in database-related decision gateways. Then, they discuss the soundness of the well-formed BPMN models without introducing verification into practice, which is postponed as future work. The authors propose a new local semantics for the inclusive join gateway where they allow paths synchronisation. However, the latter does not represent the real semantics for this element. Last, no communication models or time elements are discussed.

In [129], the authors describe the formal specification of a *Time and Resource-Sensitive simple Business process (TRSBP)*. A *TRSBP* consists of a finite set of finite sequences of activities with timing and resource constraints. It is simple because it does not explicitly have any complex control structures, such as loops. They formalise *TRSBPs* as a *Round-based model* and describe how to specify the formal *TRSBP* semantics in the *Maude* and the *Alloy* specification languages. The authors demonstrated that the business processes described in *TRSBP* can be effectively analysed with *Maude* and *Alloy*, and they demonstrate the feasibility to achieve formal analysis of *TRSBPs* with *Alloy*, while it is not with *Maude*. This is shared with our experience that business processes can be effectively analysed with *SAT-based bounded* model checking using *Alloy* language.

In [130–132], the authors present a verification tool based on the Eclipse IDE, called *Business Process Verifier (BProVe)*. BProVe allows one to verify relevant properties for business process models automatically. It is based on the translation of a subset of BPMN models into *LTS*, with a native semantics provided in [133], defined according to a *Structural Operational Semantics (SOS)* style [134]. Their proposed operational semantics is implemented using Maude to use the *Maude Linear Temporal Logic (LTL)* model-checker to verify properties and support the verification of safety and soundness properties. In [135] the authors focus on a particular element of BPMN which is the OR-join gateway. They propose formal semantics for the latter in terms of local and global views. In [136], the authors integrate their framework into Apromore [137] in the form of a plugin for the Editor environment. The authors limit the subset of supported elements and leave out aspects and constructs such as timed events, data objects, subprocesses, error handling, and multiple instances. They focus on the BPMN subset, including communication elements (send and receive), tasks and events. However, they don't give semantics for the communication model used. In [138], the authors extend their formal framework to include multiple instances and data perspectives. They also provide an associated model animator, called *MIDA*. It may help the process designers to visualise the behaviour of their models and debug them. In [139], the authors extend their operational semantics in term of *LTS* to support the verification of the hierarchical processes, including subprocesses. They provide a framework, based on their operational semantics, called *S3*. As far as verification is concerned, the authors support an extended definition of safeness and soundness properties for collaboration diagrams, taking into account distinctive characteristics introduced by message exchanges and subprocess elements. They introduce a new property, called message relaxed soundness. These properties are defined in [140]. In [141], the authors extend their formal operational semantics to cover choreography diagrams and check the conformance degree between these choreography diagrams and the collaboration diagrams by generating their LTS. Then, they perform bi-simulation-based and trace-based conformance checks.

In [142], the authors propose an approach for the verification of BPMN models with time features. First, the authors provide an encoding of the execution semantics of a subset of time-enriched BPMN elements using rewriting-logic encoded in *Maude*. Next, they show how could perform real-time analysis of such BPMN processes. Specifically, they use simulations, reachability analysis and model checking, and calculate specific properties such as minimum and maximum expected response times, the maximum degree of parallelism, and synchronisation times using *LTL* model checking also with *Maude*. Finally, they evaluate their approach on an example using a prototype tool, called *BPMN-MAUDE*. The authors focus on the time aspects, looping behaviours (generated by split and join successive gateways), and inclusive gateways with extended semantics distinct from the one given in the BPMN 2.0 standard.

In [143], the authors extend their rewriting logic executable specification of BPMN with time and probabilities supporting the automatic analysis of stochastic properties via statistical model checking. They extend the BPMN model elements with duration times and delays for tasks, flows, and gateways branching specified with stochastic expressions. The authors provide an extended semantics for the behaviour of the BPMN elements (*e.g.*, timeouts for tasks and stochastic delays for gateways paths and local behaviour for inclusive gateways as in the BPMN 1.0 version.) which may give a meaning distinct from the expected execution by using industrial modelling tools. The authors extend their work in [144, 145] to support a subset of collaboration diagram elements, resources and the multiple executions of a process. The authors extended the BPMN specification element by associating time with flows, tasks, and rates for exclusive and inclusive split gateways. They present resources as a set of identifies associated with a quantity for each of them. The authors implement their approach in *MAUDE* to stochastically simulate multiple concurrent executions of a process instance that compete for the shared resources. Then, they perform automatic verification for a set of resource allocation properties (*e.g.*, resource charge over time and usage percentage for each resource replica.). The proposed approach is illustrated with several examples using a prototype, called *BPMN-R*. These papers only address the collaboration models with a small subset of elements (massage catch event and send task) without any communication model details.

In [146], the authors provide a symbolic executable rewriting logic semantics of BPMN using the rewriting modulo satisfiability modulo theories framework, called *BPMN-SMT*. The provided semantics is based on an enhanced extension of BPMN representation supporting conditions and data flow. They associate expressions and assignments operations to tasks and gateways branches, yet without using data objects elements. The authors use rewriting modulo axioms for driving the execution and rely on *Satisfiability Modulo Theories (SMT)* decision procedures for data conditions. For property checking, the authors use Maude's rewriting logic framework, focusing on deadlock freedom and the detection of unreachable states.

In [147], the authors define an encoding for a subset of BPMN elements into the *Prolog* declarative language. They focus on a small subset of BPMN control flow elements covering a process model, extending with input, output and internal data processing. The authors analyse process models regarding external and internal consistency requirements. The external requirement covers the structure correctness of business processes. In contrast, the internal requirement covers the structure correctness of the local structure correctness of the process models and correctness criteria such as deadlock freedom, termination and determination.

In [148, 149], the authors propose an operational semantics for time-aware business processes. The authors propose an extension of the BPMN with duration annotations, defined as constraints over integer numbers. Then, they provide an operational semantics for the extended BPMN with respect to the temporal properties to be verified as a set of *Constrained Horn clauses (CHCs)*. The authors use *CHCs* solvers (Eldarica and Z3) to check the satisfiability of such clauses. Then, they use the *VeriMAP* transformation system to translate the *CHCs* into the *SMT-LIB* language, in which an *SMT* solver is invoked to check for properties satisfiability. As far as verification properties are concerned, the authors focus on weak controllability and strong controllability that guarantee, in two different senses, that all process tasks can be completed, satisfying the given duration constraints, for all possible values of the uncontrollable durations.

In [150], the authors propose an approach to use the *Alvis* modelling language for the formal analysis of BPMN models. The authors focus on the control flow elements of BPMN, taking into account or-joins as well as multiple joins, split conditions and the interaction with external participants (presented as black box pools in BPMN models). They provide a transformation of a BPMN model into an *Alvis* model. This mapping allows one to perform the formal verification of the BPMN models by generating automatically *LTS* from the *Alvis* models. These graphs can then be analysed based using the *CADP* Tool.

In [151], the authors propose a transformation rule of the BPMN model into a formal model written in Event-B. For the verification purpose, the authors use *Rodin* platform as a theorem prover for consistency conditions on BPMN models. The authors cover a large set of BPMN elements, including comprehensive modelling of control flows, data modelling, compensation, message-based communication, error and exception handling, subprocesses, looping and multi-instance activities.

In [152], the authors propose a model-driven framework that transforms a BPMN specification into a formal specification using the Event B notation. The authors use a meta-model transformation which is based on a set of mapping rules that translate the concepts of a *meta-model source (BPMN)* to a *meta-model target (Event-B)*. They automate this transformation by implementing a prototype tool called *BPMN2EventB*. For verification purposes, the authors express properties of the system in *Event-B* and check them using the *Rodin* prover. However, no information is given about the set of BPMN elements covered by the transformation nor the properties which can be checked.

In [153], the authors propose a methodology for defining process models using the *Z3* solver to verify properties by considering both syntactic and structural aspects. First, the authors introduce a meta-model of a subset of BPMN elements that covers the control flow elements for modelling a business process. Then, the meta-model of the BPMN subset is formally specified in *Z* specification as state-space schema to provide a generic approach in which any modelled process can be formally specified to check its syntactic and structural correctness. Next, the formal specification that has been defined is given for the selected modelling elements and their associated syntactic and structural rules. Then, the authors propose a manual translation between Z specification and Z3 solver for performing the verification. Finally, the authors validate their on a simple business process model and verify the control flow properties (deadlock, reachability, and dead task) using *Z3*.

In [154], the authors propose *SMT* based approach for formalising a data-aware extension of the BPMN standard. First, they describe a new formalism for representing read-only database schemata towards verifying integrated models of processes and data. Then, they support parameterised verification of safety properties of DABs using the *Model Checker Modulo Theories (MCMT)* model checker. However, no systematic evaluation is given in the paper.

3.3.5 Approach based on a Programming Language

Another complementary approach used programming languages and deduced the correctness of the model from the produced functional specifications.

In [155], the authors propose a Java-based verification approach for BPMN 2.0 collaboration models. They provide a precise mapping for each element of the BPMN notation to *Java code*. Their approach is given as an algorithm supported by a plug-in for the Eclipse IDE, called *COWSLIP*. Their approach allows the verification of the deadlock and the livelock properties based on the execution trace tree identification. It suffers from the BP integration problems in inter-organisational BP due to the undefined communication protocol used to interact with the processes. This forces the authors to ignore deadlocks related to communication. However, this is not really a resolution as communication is the heart of the collaborative correction lands.

Table 3.2: *Languages used Among the Selected Work.*

	Language	Description	Works
Petri nets	Petri Nets	is a mathematical formalism for modelling, specifying, simulating, and verifying of distributed systems.	[72], [75–77], [78], [80], [64],[67],[87]
	Workflow-Net	is a subclass of PN that is used to model the work-flow of process activities.	[72], [76, 77]
	Timed Petri Nets	is an extension of PN with a timing interval constraint on each transition.	[99], [81], [82], [87], [91]
	Coloured Petri Nets	is an extension of PN with coloured tokens well-suited for systems in which communication, Synchronisation and resource sharing are important.	[68, 69, 83, 84, 89, 90]
	Hierarchical Coloured Petri Nets	is an extension of CPN with the possibility of substitution transitions to subnets.	[92]
	Timed Petri Nets	is an extension Petri nets by associating a firing finite duration to each transition in the net.	[91], [87],[82], [81]
	RECATNets	is an extension of PN introduced to model systems with dynamic structures.	[85, 86]
	ATL	is a model transformation language provides ways to produce a set of target models from a set of source models.	[85, 86]
	YAWL Net	is a language with a strictly defined execution semantics inspired by Petri Nets.	[94–96]
	Graph Rewriting	concerns the technique of creating a new graph out of an original graph algorithmically.	[70, 71]
Automata	Finite State Machines	is a behavioural with a finite number of states model that is used to model logic.	[105]
	Timed Automata	is a finite state machine annotated with conditions and a finite set of clocks.	[101, 102], [104], [105],[106]
	Stochastic Automata Network	is a number of individual stochastic automata which is an extension of a nondeterministic finite-state machine with probabilities on the transitions.	[100]
	Kripke Structure	is a variation of nondeterministic automaton used in model checking to represent the behavior of a system.	[97, 98], [110]
	Büchi automaton	is a Finite State Machines which accepts infinite inputs (words).	[111]
	Markov Decision Processes	is a mathematical framework for modelling decision making in discrete, stochastic, and sequential environments.	[120–122]
	Promela	is a verification modelling language, allows for the dynamic creation of concurrent processes to model. It allows defining the communication via message channels to be synchronous or asynchronous	[69]
	Generic Property Specification Language	is a property-specification language that supports Linear Temporal Logic and Büchi Automata specifications.	[111]
Process Algebra	COWS	is a foundational calculus for web services.	[118]
	LOTOS New Technology	is a formal specification language, defined based on a combination of process calculi, functional languages, and imperative languages, used to specify and verify concurrent systems.	[107, 109]
	Communicating Sequential Processes	is a formal language, based on message passing via channels, for describing patterns of interaction in concurrent systems.	[114–117]
	Communicating Sequential Processes+Time	is a real time specification language that extends CSP by allowing the description of complex event timings within a single sequential process.	[119]
	Discrete Event System Specification	is a modular and hierarchical formalism for modelling and analysing complex dynamic systems using a discrete-event systems.	[123]
	NUSMV	is a specification language for describing and verifying deterministic and non deterministic systems expressed mathematically as finite state systems. It allows for the definitions of bounded arrays of basic data types.	[124]
Logic Formula	Rewriting Logic	is a logical framework for the specification of languages and systems. It supports efficient equational reasoning and specification, verification, and programming.	[128],[85],[86],[133], [131],[132]
	Maude	is a declarative language based on rewriting logic.	[129], [130–133, 135, 136, 138–141] [142–146], [128]
	Linear Temporal Logic	is a modal temporal logic with modalities referring to time.	[85, 86, 125], [110], [126]
	Prolog	is a logic programming language, well-suited for specific tasks that benefit from rule-based logical queries as it expressed programs as terms of relations.	[147]
	Horn Clauses	are a Turing-complete subset of predicate logic, allow expression parallel, concurrent, as well as sequential execution.	[148, 149]
	Event B	is a formal specification, allows modelling and system refinement at different abstraction levels, and system analysis by the use of mathematical proof to verify consistency between the refinement levels.	[151, 152]
	Satisfiability Modulo Theories	is about checking the satisfiability of logical formulas over one or more theories expressed in classical first-order logic with equality.	[146], [154], [148, 149],[153]
	Z3	is a theorem prover based on satisfiability modulo theories.	[148, 149, 153]
	Z Specification	is a model oriented formal specification language based on Zermelo-Fränkel axiomatic set theory and first order predicate logic. It is used for describing and modelling computing systems.	[153]
	Round-based model	is used to formalize distributed algorithms. It is a semi-synchronous state machine in which it takes one unit of time to complete one round. It presents common message-passing models.	[129]
	Labeled Transition Systems	is a form of abstract machine used to model one or more computations.	[107–109], [139, 141],[150]
	Property Sequence Chart	is a language that extends a set of UML2.0 interaction sequence diagrams.	[111]
	Generic Property Specification Language	is the language used by OBP2 for specifying the properties that should be verified during the analysis.	[111]
	Alvis	is a modelling language combines formal methods and practical modelling languages. Alvis modelling environment creates in parallel a model of the considered system and a labelled transition system (LTS graph) that is its formal representation.	[150]
	Time and Resource-Sensitive simple Business process	Consists of a finite set of finite series of activities that have timing and resource constraints.	[129]
	Computation Tree Logic	It is a branching-time logic, defines a model of time as a tree-like structure in which the future is not determined.	[80],[124]
	Probabilistic Control Tree Logic	is a temporal logic that allows for probabilistic quantification of system's specification properties.	[120–122]
	Clocked Computation Tree Logic	s a propositional temporal logic that extends Computation Tree Logic (CTL) with quantitative time bounds for expressing real time properties.	[101, 102]
	Timed Computation Tree Logic	an extension of CTL logic, where the bound of a temporal operator is given as a pair: a lower bound and an upper bound.	[106]
	Structural Operational Semantics	is a category of formal programming language semantics in which certain desired properties of a program are verified by constructing proofs from logical statements about its execution and procedures.	[130–133]
Others	Java	is a programming language.	[155]
	XSLT	A language for transforming XML documents into other XML documents, or other formats.	[91]
	BPEL4WS	is a language for the formal specification of business processes and business interaction protocols.	[68]
	XSLT	is a language for transforming XML documents into other XML documents, or other formats such as HTML, plain text, or XSL.	[91]
	XML	is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.	[94–96]
	XMI	is an Object Management Group (OMG) standard for exchanging metadata information via XML.	[72], [75]

Table 3.3: *Technologies used Among the Works.*

Tool	Description	Works used these technologies
Alvis	A formal modelling language and verification, tool set, and framework for model checking distributed concurrent systems.	[150]
BTransformer	A transformation java plugin for BPMN models to two outputs: CSP+T models using the sub-option BPMN2CSPT or TA network using the sub-option BPMN2TA.	[101, 102]
CPN	A tool for editing, simulating, and analysing Coloured Petri nets	[68], [83, 84]
CADP	A tool for the design, simulation and verification of asynchronous concurrent systems.	[107],[109]
DEVS-Suite platform	A modelling and simulation environment allows to construct and experiment with dynamic models based on web technology.	[123]
FDR	A refinement checking tool for models expressed in Communicating Sequential Processes (CSP) theory.	[114–117, 119]
GpenSIM	A tool for modelling, simulation, performance evaluation, and control of discrete-event systems.	[87]
GrGen.Net	A programming productivity tool for graph transformation, offers declarative languages for graph modelling, pattern matching and rewriting, as well as rule control for developing at a natural level of abstraction graph-based representations.	[70, 71]
GROOVE	A graph transformation tool is used as a modelling formalism on top of which a model checking is built, resulting graph transformation systems, allows to verify model transformation and dynamic semantics through an (automatic) analysis.	[83, 84], [126]
Lola	A tool for the verification of Petri nets models.	[72], [75], [78], [80]
Maude	A system supporting both equational and rewriting logic computation for a wide range of applications, including development of theorem proving tools, language prototyping, executable specification and analysis of concurrent and distributed systems.	[128],[142],[129] [130–132, 139]
MCMT	A model checker for infinite state systems based on the integration of SMT solving and backward reachability.	[154]
NuSMV	A symbolic model checking tool based on Binary Decision Diagrams for temporal logic.	[105], [124] ,[110]
OBP	A requirement verification environment, allows debugging, simulating and model checking the Domain-Specific Languages (DSLs) , it defines a GPSL language for specifying the properties that should be verified during the analysis and enables the integration of domain specific formalisms like UML, BPMN, TLA+, Fiacre.	[111]
ProM	A tool performs conformance checking of a system given as Petri net.	[64]
PRISM	A tool for the modelling, analysis and formal verification of systems that exhibit probabilistic behaviour.	[121], [122], [118]
PVeStA	A statistical model checking supports statistical verification and quantitative analysis of probabilistic systems.	[143]
Prolog	A Prolog compiler with constraint solving over finite domains.	[147]
PEPS	A software tool packages developed to solve and analyse Stochastic Automata Networks (SAN) models, which represents the underlying Markov chain in a compact tensor algebra format.	[100]
Rodin	A platform empowered by a large number of plug-ins providing various analysis specialised provers, capabilities like model checking, and simulation.	[151],[152]
SPIN	A verification tool for multi-threaded applications.	[69], [97, 98]
TINA	A tool for the editing and analysis of timed Petri nets	[87], [82]
UPPAAL	A tool for modelling, simulation and verification of real-time systems.	[99],[101, 102],[104],[106]
Woflan	A diagnostic tool performs the analyses of Workflow processes specified in term of Petri nets.	[72],[76],[77]
Z3	A Solver tool for deciding the satisfiability of formulas writing in a Satisfiability Modulo Theories (SMT)	[148, 149],[153]

3.4 Discussion

In this section, we discuss the related work with reference to our survey questions 3.2.2.

- Which formal model languages are used to formalise the semantics of BPMN ?

  We identified a total of 79 works that present formal frameworks, methods, methodologies, algorithms, or plug-ins for BPMN verification. We have classified them into five groups based on the formalisation model: *Petri nets, automata, process algebras, logic formulas, and programming languages.* Table 3.4 gives a summary of the research contributions. It presents the input and the output language, the used verification or simulation tool, the verification method, and the correctness properties addressed if they exist. The table shows that most of the works are based on transformation techniques rather than direct semantics. Even though such direct approaches exist, based on logic formulas, they are not generic. *i.e.*, each new work tries to give a formal semantics according to its destination language, and it does not take into account the semantics provided by the previous works. The latter is due to their chosen formalisation language, which was not interoperable.

- What are the goal(s) of this formalisation ?

  Business process verification is the act of determining if a business process model is correct concerning a set of formal properties. According to the literature, this verification in the context of business process modelling may have different goals:

  Properties verification. The correctness of information systems are related to the correctness of their behavioural properties. The behavioural properties of such systems are mainly classified into safety, liveness and soundness. A safety property denotes that nothing bad will happen, ever, during the execution of a system. Liveness denotes that something good will happen, eventually, during the execution of a system. The soundness property denotes the validity of a system. The formalisation of the execution semantics provides a mathematical basis for validating such properties with respect to a system specification. Among the works we surveyed which have this objective, we can cite [64, 67] which present the first work define the soundness, the deadlock and the liveness of BPMN models in the sense of workflows, and [139] which determine the safety and the soundness of the BPMN models in terms of collaboration.

  Business compliance. This ensures that business process models are following relevant compliance requirements. Compliance is interested in whether process models conform to specifications, which can be another process model or a set of rules, such as (inter)national laws and standards. Thus, compliance verification does not aim to prove the correctness of the business process itself but merely whether it adheres to a set of rules. Among the works we surveyed which have this objective, we find [97] that focuses on the compliance checking rules for the verification of BPMN process model soundness, and [102] that focuses on the verification of business process tasks constrained by a set of business rules.

  Business process models variability. Variability indicates that parts of a business process are variable or not fully defined to support different versions of the same process depending on the intended use or execution context. BPM variability aims to reduce development and maintenance efforts and check BP behaviours over a set of conditions. Among the works we surveyed, we find [106] which has for objective equivalence checking between processes.

  Compatibility between processes. The compatibility between processes is to compare BPMN diagrams and assert correctness conditions based on a set of defined patterns. Among the works mentioned which have this objective, we can cite [114], which addresses compatibility checking between the participants in business collaboration.

- Which is the state of tool support ?

  We have identified 26 works supported by a tool, whether a framework, a prototype or a plug-in. Four of them do not mention any availability link, and three others give one, but the given link is dead. We can then compare the 19 others. Table 3.5 presents the set of tools extracted

Table 3.4: *Synthesis of Verification Related Attributes. (Petri Nets (PN), Automata (A), Process Algebras (PA), Logic Formulas (LF), and Programming Language (PL))*

	Work	Input Language	Year	Output Language	Verification Tool	Method	Properties
PN	[64, 67]	BPMN 1.0	2008	PN	ProM	Model Checking	Soundness, Deadlock, and Livelock
	[94–96]	BPMN 1.0	2008-2010	YAWL net	WofYAWL Tool	Model Checking	Deadlock, No Dead Task, Proper completion, No OR-join, and Soundness
	[68]	BPMN 1.0	2008	CPN	CPN tool	Model Checking	Deadlock and Infinite loop
	[69]	BPMN 1.0	2009	CPN	SPIN	Model Checking	Security
	[70, 71]	BPMN 1.0 - 2.0	2010-2013	Graph rewriting	GrGen	Simulation	Conformance checking
	[72, 75]	BPMN 1.0 - 2.0	2011-2013	PN	LOLA	Model Checking	Soundness
	[76, 77]	BPMN 2.0	2014-2016	PN & WF-net	Woflan	Model Checking	Soundness and Diagnostic errors
	[82]	BPMN 2.0	2016	TPNs	TINA	Bounded Prioritised Model Checking	Liveness and Reachability
	[83, 84]	BPMN 2.0	2016-2020	CPN	CPN tool	Model Checking	Safety and Vivacity
	[85, 86]	BPMN2.0	2016-2017	RECATNets	MAUDE	LTL Model Checking	Process Termination and Deadlock
	[89]	BPMN 2.0	2018	CPN	CPN tool	Model Checking	Deadlock and livelock
	[81]	BPMN 2.0.2	2019	TPNs	TINA toolbox	Model Checking	Soundness
	[91]	BPMN 2.0	2018	TPNs	TINA toolbox	Model Checking	Deadlock and Dead-Process (timeout)
	[87]	BPMN 2.0	2018	PN	TINA & GPenSIM	Model Checking and Simulation	Generic PN properties
	[78]	BPMN 2.0	2019	PN	LOLA	Model Checking	Anti-patterns properties
	[80]	BPMN 2.0	2020	PN	LOLA	Model Checking	Data-value centred properties
	[92]	BPMN 2.0.2	2020	HCPN	CPN Tool	Model Checking	K-soundness
A	[99]	BPMN 2.0	2011	TA	UPPAAL	Model Checking	Deadlocks and Bottlenecks
	[100]	BPMN 2.0	2011	SAN	PEPS	Solving Markov Chain	Performance Evaluation
	[105]	BPMN 2.0	2016	FSMs	NuSMV	Model Checking	Business Rule properties
	[110]	BPMN 2.0	2018	KS	NuSMV	Model Checking	Refinement Safety property
	[97, 98]	BPMN 2.0	2013	KS	SPIN	Model Checking	Deadlocks, livelocks, and Multiple Terminations
	[104]	BPMN 2.0	2014	TA	UPPAAL	Model Checking	Deadlock freedom & Process deadline properties
	[101, 102]	BPMN 2.0	2014	TA	UPPAAL	Model Checking	Specific properties for a study case
	[106]	BPMN 2.0	2015	TA	UPPAAL	Model Checking	Interoperability Requirements
	[109]	BPMN 2.0	2017	LTS and LNT	CADP	Model Checking	Equivalence Checking, Deadlock, Livelock, Safety, and Liveness
	[111]	BPMN 2.0	2020	BA	OBP	Model Checking	Specific properties for a study case
PA	[118]	BPMN 1.0	2007	COWS	PRISM	Stochastic Model Checking	Quantitative properties
	[114–117]	BPMN 1.0	2008	CSP	FDR	Consistency checking	Processes Compatibility
	[119]	BPMN 2.0	2012	CSP+T	FDR2	Parallel Refinement Model Checking	Safety and Liveness
	[120–122]	BPMN2.0	2012-2016	MDP	PRISM	Stochastic Model Checking	Quantitative Safety
	[123]	BPMN 2.0	2020	JAVADEVS	DEVS-Suite platform	Simulation	Intra-activity, Inter-activity, and Inter-processes time constraints
LF	[124]	BPMN 1.0	2010	NuSMV	NuSMV	Model Checking	Specific properties for a study case
	[151]	BPMN 2.0 (BETA 1)	2010	Event B	Rodin Platform	Theorem Proving	Consistency Conditions on a study case
	[125]	BPMN 1.2	2012	LTL	×	×	Deadlock and Liveness properties
	[147]	BPMN 2.0	2012	Prolog	PROLOG	Abstract Interpretation	Local and Global correctness requirements
	[128]	BPMN 2.0	2014	RL	MAUDE	LTL Model Checking	Soundness, Liveness and Complete-path
	[150]	BPMN 2.0.2	2017	LTS	CADP	Model Checking	Specific properties for a case study
	[130–132]	BPMN 2.0	2015-2018	SOS	MAUDE	LTL Model Checking	Soundness and Safeness
	[139]	BPMN2.0.2	2020	LTS	MAUDE	LTL Model Checking	Safeness, Soundness, and Message-relaxed soundness
	[142]	BPMN 2.0.2	2017	RL	MAUDE	LTL Model Checking	Safety, Liveness, and Time Processing
	[143]	BPMN 2.0.2	2018	PRL	PMAUDE and PVeStA	Statistical Model Checking	Time properties
	[144, 145]	BPMN 2.0.2	2018-2019	RL	MAUDE	Statistical Model Checking	Time Processing and Resource based properties
	[146]	BPMN 2.0.2	2018	RL & SMT	MAUDE & SMT Solver	Bounded Model Checking	Reachability properties
	[129]	TR-SBP	2016	RL & FOL	MAUDE & Alloy	Bounded Model Checking	Specific properties for a study case
	[148, 149]	BPMN2.0.2	2016-2019	CHC	CHC Solver	Theorem proving	Weak and Strong Controllability
	[152, 156]	BPMN 2.0.2	2019	Event B	Rodin Platform	Theorem Proving	Deadlock and Proper Completion
	[154]	BPMN 2.0.2	2019	SMT	MCMT	SMT Model Checking	Safety
	[153]	BPMN 2.0.2	2020	Z	Z3	Theorem Proving	Deadlock, Reachability, and Dead Task
PL	[155]	BPMN 2.0	2012	JAVA	Eclipse	Unfolding Algorithm	Deadlock and Livelock

from the work we surveyed. For each tool, we give its underlying formal model/language support, link, objective, and supporting analysis type. Table 3.5 shows 22/26 tools for the formalisation of BPMN with a primary goal of verification and different perspectives sighted: works aim at the analysis of the behaviour of activities and their hierarchical characteristics, of message exchanges and collaboration between processes, of process in the presence of temporal elements; works aim at the analysis of execution time; works aim at quantitative analysis for the evaluation of the performance of models; works aim at security analysis. Despite the different desired goals, the primary way to reach those goals is to have a formal model for the semantics of BPMN.

- Which are the parts of BPMN being supported by the tools ?

  Table 3.6 shows the BPMN features elements supported in each identified tool. The table shows that the most addressed elements are the control flow (tasks, gateways –parallel, exclusive, and inclusive–, start and end events). Looking at this subset, we find that 90% of the works support inclusive gateways with simplified semantics marked as local in the table. Only the work presented in [71] has taken into consideration the global semantics of this gateway. On the other hand, the table shows that the communication elements are less supported, and even if found, the works focus on the send and receive tasks only and ignore the other events. Finally, the table shows that the less supported elements are data and time perspectives.

- What are the challenges that still need to be addressed ?

  Table 3.7 describes some limitations in the existing approaches. In order to give a general response to the question, we collect the limitations presented in Table 3.7 in six points as follows:

  - **Lack of a Generic Formal model.** We noticed that most of the approaches handle the formalisation of the BPMN semantics using a transformation approach to a formal modelling language such as Petri nets or timed automata. The availability of dedicated verification tools is possibly the main reason for such a choice. However, this kind of formalisation suffers the typical problems introduced by mapping into another model, where the formal semantics of BPMN is not given in terms of its mathematical definition but rather as transformation patterns to be assembled. Further, it is restricted to the semantic features of the target model, e.g., in the collaboration diagrams, tokens may be only on the sequence edges of the involved processes, while in its Petri nets model translation, the tokens refer to both messages and sequence edges. Such a distinction is not considered in the proposed translations because a message is rendered as a (standard) token in a place. Other approaches are given a direct formalisation in an encoding language such as Java or Maude. In such a case, the semantics provided by the translation is related to the low-level details of the encoding, possibly distinct in abstraction to the features and constructs of BPMN. This may make the verification results inaccurate since translations usually rely on their target language-related abstractions. Despite the different languages used in the literature, we find that the lack of expressiveness of these languages makes their use as a basis for a future extension of work complicated than redefinitions
  - **Lack of Parametric Communication Support.** Several works support the semantics of collaboration-related elements (collaboration diagram, sending and receiving messages). However, they perform verification based on synchronous communication models. None of these works has studied the verification of BPMN models under different configurable interaction/communication models.
  - **Lack of Timed Elements Support.** Several works address the Timed perspective of BPMN. However, all these works focus on extending the BPMN notion to support the time constraints. No identified work has discussed the semantics of timer elements regarding the ISO-8601 standard as specified in the BPMN standard [3, P 274, Chapter 10].
  - **Limited subset of BPMN being supported.** Complex elements such as OR join gateways, subprocesses, timer events, and the collaborations between processes, multi-instance characteristic is well supported in some works only. However, none of these works can reason on collaboration, including the semantics of all of these elements at the same time.
  - **BPMN Correctness Properties.** Concerning verification, different properties have been defined in Petri Nets, timed automata, process calculi and other formal languages. The most known ones in the BP context are soundness and safety, defined in [177]. The soundness

Table 3.5: *Synthesis of Tools.*

Approach	Reference	Formalism	Tool		Objective	Analysis		
			Name	Link		Simulation	Verification	Animation
Mapping	[64]	PN	Transformer	[65]	Enabling analysis and verification for BPMN models using Petri net-based semantics.	×	✓	×
	[69]	CPN	Oryx	[157]	Verifying the control access security properties defined on business process model.	✓	✓	×
	[97, 98]	KS	EPSPIN	[158]	Automatic checking of structural errors.	×	✓	×
	[101, 102]	TA	BPMN2TA	×	Supporting the analysis of temporal business properties and rules in BPMN diagrams based on their associated BP–task models as a TA networks.	×	✓	×
	[111]	BA	OBP2	[112]	Performing static syntax checks and semantics execution analysis (interactive model animation, trace simulation, and properties verification) with OBP technology.	✓	✓	✓
	[71]	Rewriting Graph	GrGen.NET	[159]	Formalising and visual debugging of BPMN models based on graph transformation rules.	✓	×	×
	[86]	RECATNETS	BPMN Checker	[160]	Supporting the verification of complex collaborative business processes models with data flow, control flow, multiple instances, and exception handling addressed using meta-modelling between BPMN and RECATNET.	×	✓	×
	[94–96]	YAWL	BPMN2YAWL	[161]	Deployment and analysis of BPMN models with YAWL workflow language and its verification tools.	×	✓	×
	[100]	SAN	BP2SAN	[162]	Analytical performance evaluation of business process models with SAN models.	✓	×	×
	[120, 121]	MDP	SBAT	×	Performing quantitative probabilistic model checking of BPMN process diagrams.	×	✓	×
	[89]	CPN	CPN4BPMN	×	Performing the verification of structured and unstructured BPMN models in control flow and data flow contexts based on a refinement approach.	×	✓	×
	[92]	HCPN	fcm2cpn	[93]	Performing the translating of BPMN process fragments to CPN	×	✓	×
Direct	[114]	CSP	Machine-readable CSP	[163]	Formally analyse and compare BPMN diagrams.	×	✓	×
	[155]	Java	Cowslip	[164]	Adaptation of an unfolding exploration technique to support BPMN verification based on Java models.	✓	✓	×
	[119]	CSP+T	BPMN2CSPT	×	Formalising and verifying the behavioural and temporal aspects of BPMN models extends with temporal constructs.	×	✓	×
	[131–133]	LTS & Maude	BPROVE	[165]	Support a formal verification of BPMN collaboration diagrams.	✓	✓	×
	[138]	LTS	MIDA	[166]	Animating BPMN models in collaborative, multi-instance and data-based contexts	×	×	✓
	[141]	LTS	C4	[167]	Conformance checking of collaborations w.r.t. choreographies	×	×	×
	[139]	LTS	S3	[168]	Verifying BPMN collaboration diagrams in message exchange and/or sub-processes context	×	✓	×
	[109]	LNT & LTS	VBPMN	[169]	Checking properties of BPMN business process diagrams using the model checker CADP.	×	✓	×
	[142]	Maude	MAUDE-BPMN	[170]	Performing real-time analysis (measures of execution time (minimum, maximum, average) or the degree of parallelism) on BPMN models	×	✓	×
	[143]	Maude	BPMN-P	[171]	Verification of stochastic properties on BPMN models extended with probabilistic specification of time and branching constructs.	×	✓	×
	[144, 145]	Maude	BPMN-R	[172]	Verification technique for identifying, optimising, analysing the allocation of resources in business processes.	×	✓	×
	[173]	SMT	BPMN-SMT	[174]	Symbolic analysis and verification of business process models with data support.	×	✓	×
	[152, 156]	Event B	BPMN2EventB	[175]	Supporting the formalisation and verification of business process models based on meta-modelling and model transformation approach to avoid state number explosion and correctness validation of the transformation.	×	✓	×
	[149]	Horn Clauses	VeriMAP	[176]	Verifying the controllability of time-aware business Process models	×	✓	×

Table 3.6: *Embraced BPMN Features (• stands for the supported elements, −− for the non-supported elements, (•) mentions that is not clear).*

BPMN 2.0 objects	[64]	[69]	[101, 102]	[97, 98]	[111]	[71]	[86]	[94–96]	[100]	[120, 121]	[89]	[92]	[114]	[155]	[119]	[131–133]	[138]	[141]	[139]	[109]	[142]	[143]	[144, 145]	[173]	[152, 156]	[149]
Start Events																										
None start	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
Message start	•	−−	−−	−−	−−	•	•	−−	−−	−−	−−	−−	−−	•	−−	•	•	−−	•	−−	−−	−−	−−	−−	−−	−−
Timer start (cycle)	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
Timer start (Date)	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
Timer start (Duration)	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
Message catching	•	•	•	•	−−	•	•	•	−−	−−	•	−−	−−	−−	•	•	•	•	•	−−	−−	−−	•	−−	•	−−
Message throwing	•	•	•	•	−−	•	•	•	−−	−−	•	−−	−−	−−	•	•	•	•	•	−−	−−	−−	−−	−−	•	−−
Message boundary (Interrupt.)	−−	−−	−−	−−	−−	•	−−	•	−−	−−	•	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	(•)	−−
Message boundary (non Interrupt.)	−−	−−	−−	−−	−−	•	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	(•)	
Timer intermediate event	•	−−	•	−−	−−	−−	•	•	−−	−−	•	−−	−−	−−	•	−−	−−	−−	−−	−−	−−	−−	•	−−	−−	−−
Timer catching (Date)	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
Timer catching (Duration)	−−	−−	•	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	•	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
Timer boundary (Interrupt., Date)	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
Timer boundary (Interrupt., Duration)	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	•	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
Timer boundary (non Interrupt., Date)	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
Timer boundary (non Interrupt., Duration)	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
Timer boundary (non Interrupt., Cycle)	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
Error intermediate event	−−	−−	−−	•	−−	•	•	•	−−	−−	•	−−	•	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
End Events																										
None end	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	−−
Message end	−−	−−	−−	−−	−−	•	•	−−	−−	−−	−−	−−	−−	−−	−−	•	•	−−	−−	−−	−−	−−	−−	−−	−−	−−
Terminate end	−−	−−	−−	−−	−−	•	−−	−−	−−	−−	−−	−−	•	−−	•	•	•	−−	−−	−−	−−	−−	−−	−−	−−	−−
Error end	−−	−−	−−	−−	−−	•	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
Gateways																										
Parallel (fork/join)	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
Exclusive fork	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
Exclusive join	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
Event based join	−−	−−	−−	−−	−−	−−	•	−−	−−	−−	−−	−−	−−	−−	•	−−	−−	−−	−−	−−	•	•	•	−−	−−	−−
Event based fork	•	−−	−−	−−	−−	−−	•	•	−−	−−	•	−−	−−	−−	•	•	•	•	•	−−	•	•	•	−−	−−	−−
Inclusive fork	−−	−−	•	•	−−	•	−−	•	•	−−	•	−−	•	−−	•	•	−−	−−	−−	•	•	•	•	•	−−	−−
Inclusive join (local)	−−	−−	•	•	−−	−−	−−	•	•	−−	•	−−	•	−−	•	−−	−−	−−	−−	•	•	•	•	•	−−	−−
Inclusive join (global)	−−	−−	−−	−−	−−	•	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
Activities																										
Abstract task	•	−−	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	−−
Send task	•	•	•	−−	•	−−	•	−−	−−	•	•	−−	•	•	•	•	•	•	•	−−	−−	−−	•	−−	•	−−
Receive task	•	−−	•	−−	•	−−	•	−−	−−	•	•	−−	•	•	•	•	•	•	•	−−	−−	−−	−−	−−	•	−−
Embedded Sub-process	•	−−	−−	−−	−−	•	•	•	−−	•	−−	−−	•	−−	•	−−	−−	−−	•	−−	−−	−−	−−	−−	•	−−
Event Sub-process	−−	−−	−−	−−	−−	−−	−−	•	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
Loop task	•	−−	−−	−−	−−	•	•	•	−−	−−	•	−−	•	−−	•	−−	•	−−	−−	−−	−−	−−	−−	−−	•	−−
Loop sub-process	•	−−	−−	−−	−−	•	−−	•	−−	−−	−−	−−	•	−−	•	−−	−−	−−	−−	−−	−−	−−	−−	−−	•	−−
Multiple instance task (static)	•	−−	−−	−−	−−	−−	•	•	−−	−−	•	−−	•	−−	•	−−	•	−−	−−	−−	−−	−−	−−	−−	(•)	−−
Multiple instance task (dynamic)	•	−−	−−	−−	−−	−−	•	•	−−	−−	•	−−	•	−−	•	−−	−−	−−	−−	−−	−−	−−	−−	−−	(•)	−−
Multiple instance sub-process (static)	•	−−	−−	−−	−−	−−	•	•	−−	−−	−−	−−	•	−−	•	−−	−−	−−	−−	−−	−−	−−	−−	−−	(•)	−−
Multiple instance sub-process (dynamic)	•	−−	−−	−−	−−	−−	•	•	−−	−−	−−	−−	•	−−	•	−−	−−	−−	−−	−−	−−	−−	−−	−−	(•)	−−
Data Objects																										
Data object	−−	−−	−−	−−	−−	−−	•	−−	−−	−−	•	•	−−	−−	−−	−−	•	−−	−−	−−	−−	−−	−−	−−	−−	−−
Data store	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	•	•	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
Resources	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	•	−−	−−	−−

property consists in guaranteeing the absence of deadlock and no dead activities in the model. The safety property consists of ensuring the bounded number of tokens per sequence flow. The works mentioned above (Table 3.4) focus mainly on the verification of soundness properties related to the control flow of the business process, without considering communication aspects. In addition, the available verification tools for BPMN can not differentiate issues concerning the control flow from those concerning the message flow.

Recently, a new reformulation of the BPMN safety and soundness properties has been defined in [140] to take into account essential collaboration features of the BPMN models, such as message passing and related soundness properties (e.g., message-relaxed soundness). Taking these relevant properties into account in the verification process permits distinguishing issues concerning the control flow from those concerning the message flow. Therefore, the safeness of a BPMN collaboration only refers to the tokens on the sequence edges of the involved processes. However, such distinction is not considered in the translation approaches, e.g., in Petri Nets translation, a message is rendered as a token in a place. Hence, a safe BPMN collaboration may be considered unsafe by relying on the Petri Nets notion.

– **Lack of Empirical Evaluation.** The analysis and the evaluation of the proposals performed in most of the selected works is achieved through a case study. No benchmarks are given except for works are in Corradini. *et al* [165] and Duran *et al* [142–146]. However, Duran *et al.* takes as input extended BPMN models with extended notations and no available BPMN modelling tool to support them.

3.5 Summary

This chapter presented a systematic literature review on the formalisation of BPMN execution semantics and the annotated verification tool support. We investigated this state of the art according to five questions. We analysed 79 works spread across five distinct formal models/languages: Petri nets, timed automata, process algebras, logic formulas, and programming languages. We have given a detailed overview of each selected work. Then, we have identified their tool support (or lack of) and their empirical objectives. Finally, we have identified their limitations to lead the thesis objectives. In the following chapters, we intend to address the gaps discovered from this research by proposing formal BPMN semantics covering the process and the collaboration diagrams. Furthermore, our research will address the gaps related to communication management, time constraints, hierarchical structure supports and the (famous) OR join gateway semantics problem.

Table 3.7: *BPMN Verification Tools Limitations.*

Ref	Formal Languages	Limitations
[64]	PN → PNML	• Approach based on the version 1.1 of BPMN • According to the authors, the approach suffers from deficiencies that impact the proposed formalisation • The behaviour of the message tasks and events is not properly clear • Lack of OR-Join semantics support • No time notion is given for the timer events • Safeness notion differs on the Petri nets then when it is given directly on BPMN collaborations
[69]	CPN → Promela	• Extended Business process model with a security • Lack of OR-Join semantics support • Despite supporting communication using the intermediate message event and Promela basic building blocks are asynchronous processes with synchronous message channels. Still, no details about the communication management nor the formalisation of it is given • Difficulties with unbounded model support for the verification • Conflict on the soundness property definition with the one given at scale of business process
[101, 102]	TA	• The approach does not explicitly express. No details about the semantics of the supported set of BPMN elements • The author uses synchronous communication between the processes due to the use of TA as a formal language. Still, there is no explicit formalisation nor explication for the message exchange management, the collaborative participants, and the impact of this mode of communication on the verification • Formal representation of timing requirement is given in the work without any focus on the central semantics of the timer events and their types regarding ISO standard definitions
[97, 98]	KS	• It supports a small subset of BPMN elements reduced to the control flow elements • It gives a local semantics for the OR-join gateway • It does not respect the standard definition semantics, e.g., the message intermediate events are treated as a none task • It does not support the communication nor the timer events • It focuses on generic properties of verification and ignores the different properties that have been defined in the context of business processes
[111]	GPSL → PSC → BA	• It supports the interaction BPMN elements without any explicit formalisation for the communication • It applies three phases of model transformation without any proof that guarantees the passage from the input BPMN model to the last BA model without losing the basic concepts • According to the authors, the approach suffers from the limitation of the introduced graphical language PSC for describing properties • No support for the BP correctness properties • Lack of OR-Join semantics and the subprocess support
[71]	In-place Graph → PN	• It does not provide express semantics for the communication elements • Lack of OR-Join semantics support • Lack of the timer events support • It Does not allow to apply verification techniques
[86]	RECATNets → MAUDE	• It does not provide express semantics for the interaction elements and the their communication management • It covers the intermediate timer event elements without providing any notion of time nor supporting the ISO standard definitions • Lack of OR-Join semantics support • It does not address the business process correctness properties (safeness and soundness)
[94–96]	YAWL	• Approach based on the version 1.1 of BPMN • Lack of OR-Join semantics support • Lack of collaboration support due to the no equivalents in the YAWL notation for the pools, lanes notations, *e.g., who are the participants involved in the exchange of messages* • *Lack of timer event support with reference to ISO standard*
[100]	SAN	• Lack of collaboration models support • Lack of OR-Join semantics support • Lack of unbalanced workflows support • Small subset of supported BPMN elements • Extended all supported BPMN element with an average execution time and probabilities • It does not address the time semantics for the BPMN timer elements, but the performance evaluation of service time, waiting time, queue size, and resource utilisations and how the system workload may impact them • It does not address the verification of the BP correctness properties but improves the BP by identifying inefficiencies, such as bottlenecks and idle resources
[120, 121]	MDP → PRISM	• Supports a small subset of BPMN elements extended with probabilities • Lack of inclusive and event-based gateway support • It allows the modelling of the communication elements using send and receive tasks and maps the collaborative processes to a set of synchronised modules due to the use of synchronous PRISM actions, but it does not address the impact of the synchronisation uses w.r.t the BP correctness • It does not support the timer events • It Does not address the qualitative correctness properties, but the quantitative ones
[89]	CPN	• Lack of OR-Join semantics support • Lack of timer event semantics support • Refinement limitations regarding the deleting and adding CPN place and transition in a model • It does not support the unstructured BPMN models • According to the authors, data and control flow in the CPN model are passed along in the same arc. There is the potential for the copies of the same data to perform the same task (parallel execution). Thus, the designer should modify the task logic to choose the correct data version. • State-space generator is not sufficient for verifying the model that has a variety of the variables in a large design model • It does not address the BP correctness properties but is interested in finding Petri net properties (deadlocks and livelocks)
[92]	CPN	• It supports a small subset of BPMN elements, mainly elements using data • It does not address the collaboration nor the time perspectives • It supports compliance verification which does not aim to prove the correctness of the business process itself but merely whether it adheres to a set of rules
[114]	CSP	• Approach based on the version 1.1 of BPMN • Pi-calculus which is used for the formalisation is complex to understand • Lack of OR-Join semantics support • CSP language based on synchronous communication channels, Still, there is no details about the its impact on the correctness properties verification • It is based on the ILOG JViews BPMN Modeler which is not anymore available • It does not address the time perspective • They focus on the verification of the consistency checking, performed by using the FDR tool

[119]	CSP+T	• Extended the BPMN models with time notation, e.g., associate a time interval min and max duration for each activity • Lack of the event-based gateway semantic support • Lack of the timer intermediate event semantic support, only the duration time notion that is supported • It is evaluated only on a one study case • The authors support only the asynchronous communication mode for the message exchanges
[155]	JAVA code	• Supports a small subset of BPMN elements • The semantics given is based on the low-level details of the Java encoding as an execution traces • The authors exclude some deadlock cases for the efficiency of their approach that are necessary for the collaboration correctness, e.g., waiting process that never start due to waiting messages for start event, Deadlock caused by choice before synchronisation messages • It does not address the BP correctness properties, but it is interested in finding generic properties (deadlocks, livelocks) • It is evaluated on a simple example
[131–133]	LTS $\rightarrow$ MAUDE	• Supports a small subset of BPMN elements • Lack of OR-Join, sub-processing, time event, and data semantics support • The infinite states issue produced due to the LTSs language use • Lack of collaboration properties support
[138]	LTS	• Approach does not enable the verification
[141]	LTS	• Small subset of BPMN element support • Approach address the conformance checking between the collaboration and the chronography models and not correctness properties verification
[139]	LTS $\rightarrow$ JAVA	• It supports a small subset of BPMN elements • Lack of OR-Join, time event, and data semantics support • The authors mention the use of the asynchronous communication model for communication. But they do not introduce any details about its support nor its integration in the formal model • They implement the defined LTS semantics and the correctness checking techniques in JAVA • The proposed implementation is still a prototype that can be subject to many optimisations • The authors verify a set of BP correctness properties using the implemented tool. However, they do not express the method checking used
[109]	PIF $\rightarrow$ LNT $\rightarrow$ LTS	• It supports a small subset of BPMN elements • Lack of communication, time event, and data element support • Lack of the OR-Join semantics support • It unlikely the BPMN semantic is respected through the multiple transformations • It does not address the BP correctness properties
[142]	RL	• Supports a small subset of BPMN elements enriched with time features (e.g., timeouts for tasks and delays for branching sequence flows in gateways) to treat the time constraints, and they did not stick to the actual behaviours defined in the standard • Lack of the OR-Join and the event based gateway semantics support • Lack of communication, time event, and data element support • It does not address the verification of the BP correctness properties
[143]	Probabilistic RL	• Supports a small subset of BPMN elements enriched with stochastic expressions for specifying the time and probabilistic branching • Lack of the OR-Join and the event-based gateway semantics support • Lack of communication, time event, and data element support • It does not address the verification of the BP correctness but the quantitative performance • The probabilistic model removes relationships of activity that have small occurrences in its BP model
[144, 145]	RL	• Supports a small subset of BPMN elements enriched with stochastic expressions for expressing time constraints • Event it supports communication events, it does not address their semantics nor their verification • The authors focus on a time notion without any reference to the one given by the BPMN standard • It addresses the formal specification and verification of quantitative aspects of processes and their resources and not the qualitative ones • Approach address the stochastic simulation of multiple concurrent executions of a process instance that compete for the shared resources
[173]	symbolic RL $\rightarrow$ SMT	• Supports a small subset of BPMN elements for the modelling of a business process • Lack of the Or join semantics support • It does not address the communication nor the time perspectives in terms of the BPMN standard features
[152, 156]	Event B Mata model $\rightarrow$ RODIN	• Supports a small subset of BPMN elements without any precision (e.g., the authors mention the boundary event without any details with which event is associated message, time, or errors; is it interrupting or no, etc.) • The approach does not explicitly express. No details about the semantics of the supported set of BPMN elements, nor how they are formalised • The proposed approach is evaluated over a simple example that is not found in the folders nor the paper • Their approach allows the multi-instantiation of the process and the activities but no details about its management in the collaboration case, nor in the simple process model due of the black token uses; which type supports the dynamic one the static one
[149]	CHC	• Supports a small subset of BPMN elements extended with time constraints associated to tasks • Lack of the Or join and the subprocess elements support • Does not support the collaboration models nor the timer elements and their time perspective with reference to ISO standard definition • Event the work address the timer constraint perspective, it focuses only on the time constraints that task durations should satisfy

Part II

BPMN 2.0 Semantics Formalisation

Chapter

4 BPMN and Communication

“*Communication is the key for any global business.*”
Anita Roddick

Chapter content

4.1 Introduction

Communication is an essential human activity to represent an exchange of messages and information among people. It is also an important organisational capability used for negotiating, discussing, and making decisions about how to coordinate and cooperate in business activities. Basically, communication may be performed 'face-to-face' (*i.e.,* interpersonal communications) or remotely (*i.e.,* via a communication medium). Corporate communications have an important impact on organisational success. In line with this, communications are commonly represented by the following process: *“a sender transmits a message through a channel to the receiver”*. In this context, BPMN provides collaboration diagrams to represent communication between a set of participants. These diagrams define a precise order in which messages are sent and received. Effective communication modelling should have appropriate knowledge about the communication type on which it depends (*i.e.,* synchronous or asynchronous type of communication). Yet BPMN does not allow these communication modes to be taken into account. In addition, BPMN suffers from a lack of standard formal semantics. This weakness can lead to inconsistencies, ambiguities, and incompleteness within the developed models. As a result, many researchers proposed formal methods to build formal description and verification models of business processes. However, one of the weaknesses of these proposals is their lack of support for modelling complex BPMN collaboration business processes involving communication models. For that, we need an expressive modelling formalism that allows, on the one hand, to specify the dynamic structure of the business processes models, and on the other hand, to check the control-flow correctness of these models while taking into account their communication model.

Thus, our efforts are to define formal semantics for a relevant fragment of BPMN collaborative models, providing a modular structure for incorporating numerous possible communication models. These models are related to message-passing behaviours between and within processes and show how one can interchange them when studying a given BPMN schema. Formalising BPMN in a mathematical language would bring many advantages: (1) giving an abstract and generalised semantics, (2) being easily implementable in different verification languages, (3) being able to perform a more efficient verification of the system, and (4) being able to extend it to take into account the organisational information associated with process models such as the perspective of time constraints (cf. Chapter 5).

This chapter formalises the semantics of BPMN collaborations. It focuses on a subset of BPMN execution semantics that supports subprocesses, inclusive gateway, interaction and is parametric with respect to the communication properties. This formalisation is given in terms of First-Order Logic (FOL). This allows the translation of the process execution semantics without being linked to a particular language. The proposed semantics supports the seven point-to-point communication models [178] that exist when considering local, causal and global message ordering, and it is easily extensible. Furthermore, on top of these seven generic communication models applied to the whole collaboration, the proposal supports the definition and the use of ad-hoc communication models (a specific model built by assembling micro communication models that provide different constraints on sending and receiving messages).

This chapter is organised as follows. Section 4.2 provides the presentation of the model underlying the semantics. Section 4.3 summarises the basic concepts of the selected communication models needed for understanding this chapter and gives their formalisation in terms of FOL logic. Section 4.4 gives the formalisation of a subset of BPMN elements. Section 4.5 discuss the verification properties supported by the formalisation. A conclusion is given in Section 4.6.

4.2 A Typed Graph Representation of BPMN Collaborations Models

We formally reduce the representation of a collaboration diagram as a typed graph, where types corresponding to the BPMN element types are associated with nodes and edges. Thus, each node or edge in the graph corresponds to a BPMN node or edge. This work does not propose an alternative modelling notation, but it defines a Backus-Naur Form (BNF) syntax for BPMN models.

4.2.1 BPMN Elements Type

Figure 4.1 reports the set of BPMN elements supported in this work. In addition, it highlights the syntax defining the textual notation types of BPMN collaboration models. Types of these elements are based on the following disjoint sets:

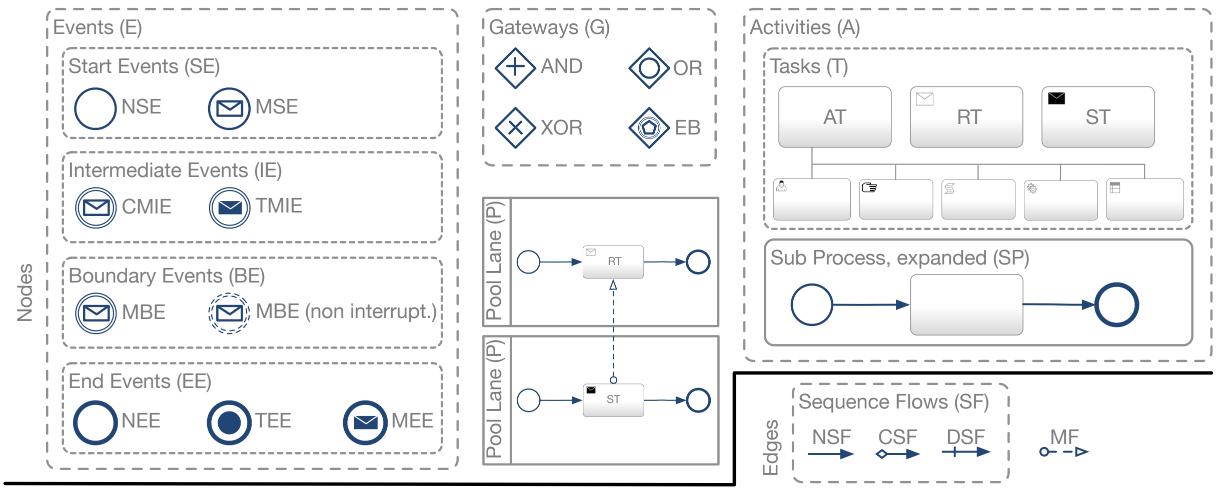


Figure 4.1: *A Subset of Supported BPMN Elements.*

- **For Nodes:**

  - The set of task (T), groups the abstract task (AT), the receive task (RT), and the send task (ST) types. Formally: $T = \{AT, RT, ST\}$.
  - The set of activity (A), groups the task and the sub-process (SP) types. Formally: $A = T \cup \{SP\}$.
  - The set of gateway (G), groups the parallel (AND), the inclusive (OR), the exclusive (XOR), and the event-based (EB) gateway types. Formally:

$$G = \{AND, OR, XOR, EB\}$$

  - The set of start event (SE), groups the none start event (NSE), the message start event (MSE), and the timer start event (TSE) types. Formally:

$$SE = \{NSE, MSE, TSE\}$$

  - The set of intermediate event (IE), groups the catch message intermediate event ($CMIE$), the throw message intermediate event ($TMIE$), and the timer intermediate catch event ($TICE$) types. Formally:

$$IE = \{CMIE, TMIE, TICE\}$$

  - The set of boundary event (BE) , groups the message boundary event (MBE) and the timer boundary event (TBE) types. Formally: $BE = \{MBE, TBE\}$. Both indeed regroup interrupting and non-interrupting versions. A function, *isInterrupt* (Def. 4.2.1), is used to make the difference.
  - The set of end event (EE), groups the none end event (NEE), the terminate end event (TEE), and the message end event (MEE) types. Formally:

$$EE = \{NEE, TEE, MEE\}$$

  - The set of event (E), is the set of all event types. Formally:

$$E = SE \cup IE \cup BE \cup EE$$

- **For Edges:**
  - The set of sequence flow (SF), groups the normal sequence flow (NSF), the conditional sequence flow (CSF), and the default sequence flow (DSF) types. Formally:

$$SF = \{NSF, CSF, DSF\}$$

  - The set of message flow (MF), is used to denote message flows.

Thus, we consider two basic disjoint sets of elements types as follows:

- T_{Nodes}denotes the set of all node types, with an added type, P, denoting processes. Formally: $T_{Nodes} = A \cup G \cup E \cup \{P\}$.
- T_{Edges} denotes the set of all edge types. Formally: $T_{Edges} = SF \cup \{MF\}$.

4.2.2 Graph Structure

After defining the sets of BPMN elements types, we introduce the notion of a collaboration diagram as a labelled graph.

Definition 4.2.1 (BPMN Graph)**.** A BPMN graph is a tuple $\widehat{G}$= (N, E, $\mathbb{M}$, cat_N, cat_E, *source*, *target*, R, msg_t, *attachedTo*, *isInterrupt*) where:

- N, is the set of nodes,
- E ($N \cap E = \emptyset$), is the set of edges,
- $\mathbb{M}$, is the set of message types,

- $cat_N : N \to T_{Nodes}$, returns the type of a node,

$$cat_N(n) \overset{def}{\equiv} \{t \in T_{Nodes} \mid \forall n \in N\}$$

- $cat_E : E \to T_{Edges}$, returns the type of an edge,

$$cat_E(e) \overset{def}{\equiv} \{t \in T_{Edges} \mid \forall e \in E\}$$

- $source : E \to N$, returns the source of an edge,

$$source(e) \overset{def}{\equiv} \{n \in N \mid e = (n, v),\ \forall v \in N\}$$

- $target : E \to N$, returns the target of an edge,

$$target(e) \overset{def}{\equiv} \{v \in N \mid e = (n, v),\ \forall n \in N\}$$

- $R : N \to 2^{N \cup E}$, returns the set of nodes and edges which are directly contained in a container (process or sub-process).

$$R(n) \overset{def}{\equiv} \begin{cases} elems \subseteq N \cup E & \text{if } \ cat_N(n) \in \{P, SP\} \\ \emptyset & \text{otherwise} \end{cases}$$

Notation. We note R^+ the transitive closure of R, and R^{-1} the inverse of R.

- *$msg_t : E \to \mathbb{M}$ returns the message associated to a message flow,*

$$msg_t(e) \overset{def}{\equiv} \begin{cases} m \in \mathbb{M} & \textit{if } \ cat_E(e) \in MF \\ \emptyset & \textit{otherwise} \end{cases}$$

- *$attachedTo : N \to N$, returns the activity to which a boundary event node is attached,*

$$attachedTo(n) \overset{def}{\equiv} \begin{cases} a \in N & \textit{if } \quad cat_N(n) \in BE \wedge cat_N(a) \in A \\ \emptyset & \textit{otherwise} \end{cases}$$

2125 - *$isInterrupt : N \to Bool$, denotes whether a boundary event node is interrupting or not,*

$$isInterrupt(n) \overset{def}{\equiv} \begin{cases} b \in Bool & \forall n \in N, cat_N(n) \in BE \\ false & \textit{otherwise} \end{cases}$$

Example of R function application. Let's take an example to show how we dealt with the hierarchical structure of the BPMN diagrams. Consider again the example of Figure 2.8 with two process node types *Customer* and *TravelAgency*. To deal with processes P and sub-processes SP containment, we use the relation, R. If we apply the R function on the *Customer* process, we obtain:

$$\begin{aligned} R(Customer) = \{&"StartTravelBooking", "RequestforOffer", "check?", "CheckTravelOffer", \\ &"istheofferinteresting?", "Informtheagency", "BookTravel", "PayTravel", \\ &"BookingConfirmed", "ReceivedInformation", "SendAbort", "Transaction \\ &Aborted", "TicketReceived", "TransactionCompleted", e_0, e_1, e_2, e_3, e_4, e_5, e_6, \\ &e_7, e_8, e_9, e_{10}, e_{11}\} \end{aligned}$$

However, the application of the function on the *TravelAgency* process returns as result only the global nodes, *i.e.*, the subprocess *OfferSP* and *ExchangeSP* without their contents.

$$\begin{aligned} R(TravelAgency) = \{&"StartOfferManagement", "OfferSP", "TimeOut", "StopSending \\ &Offer", "Continue", "ExchangeSP", "ReceiveAbord", "OfferAborted", \\ &"OfferCompleted", e_{12}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}, e_{19}, e_{20}, e_{21}, e_{22}, e_{23}, e_{24}, \\ &e_{25}, e_{26}, e_{27}, e_{28}\} \end{aligned}$$

Table 4.1: *Syntactic Representation of the Travel Agency Example.*

N =	{"Customer","TravelAgency","StartTravelBooking","RequestforOffer","check?","CheckTravelOffer","istheofferinteresting?", "Informtheagency","BookTravel","PayTravel","BookingConfirmed","ReceivedInformation","SendAbort","TransactionAborted", "TicketReceived","TransactionCompleted","StartOfferManagement","OfferSP","TimeOut","StopSendingOffer","Continue", "ExchangeSP","ReceiveAbord","OfferAborted","OfferCompleted","StartOffers","OtherOffer?","ExistMoreoffer?", "MakeTravel,Offer","SendInformation","OffersCompleted","StartBooking","BookingReceived","PaymentReceived", "ConfirmBooking","OrderTicket","BookingCompleted"}
E =	{$e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}, e_{19}, e_{20}, e_{21}, e_{22}, e_{23}, e_{24}, e_{25}, e_{26}, e_{27}, e_{28}$,"$mf_0$,"$mf_1$","$mf_2$","$mf_3$","$mf_4$", "$mf_5$","$mf_6$","$mf_7$","$mf_8$"}
$\mathbb{M}$ =	{"OfferRequest","Offer","NoMore","Abort","SelectOffer","Travel","Confirmation","Payment","Ticket"}
msg_t=	"mf_0" → "OfferRequest","mf_1" → "Offer","mf_2" → "NoMore","mf_3" → "Abort","mf_4" → "SelectOffer","mf_5" → "Travel", "mf_6" → "Payment","mf_7" → "Confirmation","mf_8" → "Ticket"
cat_N=	"Customer" → P,"TravelAgency" → P,"StartTravelBooking" → NSE,"check?" → XOR,"CheckTravelOffer" → RT, "istheofferinteresting?" → XOR,"BookTravel" → ST,"PayTravel" → ST,"TicketReceived" → CMIE, "BookingConfirmed" → CMIE,"ReceivedInformation" → BMIE,"SendAbort" → TMIE,"TransactionAborted" → NEE, "TransactionCompleted" → NEE,"StartOfferManagement" → MSE,"OfferSP" → SP,"StartOffers" → NSE, "Otheroffer?" → XOR,"ExistMoreoffer?" → XOR,"MakeTravelOffer" → RT,"SendInformation" → TMIE, "StopSendingOffer" → MBE,"OffersCompleted" → NEE,Continue → XOR,"ExchangeSP" → SP,"StartBooking" → NSE, "BookingReceived" → CMIE,"PaymentReceived" → CMIE,"ConfirmBooking" → ST,"OrderTicket" → ST,"EndBooking" → NEE, "ReceiveAbord" → MBE,"OfferAborted" → NEE,"OfferCompleted" → NEE}
cat_E=	$e_0 \to NSF, e_1 \to NSF, e_2 \to NSF, e_3 \to NSF, e_4 \to CSF, e_5 \to DSF, e_6 \to NSF, e_7 \to NSF, e_8 \to NSF, e_9 \to NSF, e_{10} \to NSF, e_{11} \to NSF,$ $e_{12} \to NSF, e_{13} \to NSF, e_{14} \to NSF, e_{15} \to NSF, e_{16} \to DSF, e_{17} \to NSF, e_{18} \to NSF, e_{19} \to NSF, e_{20} \to NSF, e_{21} \to NSF, e_{22} \to NSF,$ $e_{23} \to NSF, e_{24} \to NSF, e_{25} \to NSF, e_{26} \to NSF, e_{27} \to NSF, e_{28} \to NSF$,"$mf_1$" → MF,"$mf_2$" → MF,"$mf_3$" → MF,"$mf_4$" → MF, "$mf_5$" → MF,"$mf_6$" → MF,"$mf_7$" → MF,"$mf_8$" → MF
source =	e_0 → "StartTravelBooking", e_{12} → "StartOffreManagement" e_1 → "Requestforoffre", e_2 → "Check?", e_3 → "checkTravelOffre", e_4 → "isthisofferinteresting?", e_5 → "isthisofferinteresting?" e_6 → "BookTravel", e_7 → "PayTravel", e_8 → "TicketReceived", e_9 → "BookingConfirmed", e_{10} → "ReceivedInformation", e_{11} → "SendAbort" e_{13} → "StartOffers", e_{14} → "OtherOffer?", e_{15} → "MakeTravelOffer", e_{16} → "ExistMoreOffer", e_{17} → "SendInformation", e_{18} → "ExistMoreOffer", e_{19} → "OfferSP", e_{20} → "Continue", e_{21} → "StartBooking", e_{22} → "BookingReceived", e_{23} → "PaymentReceived", e_{24} → "ConfirmBooking", e_{25} → "OrderTicket", e_{26} → "ExchangeSP", e_{27} → "ReceiveAbord", e_{28} → "StopSendingOffer" mf_0 → "RequestforOffer", mf_1 → "MakeTravelOffer", mf_2 → "SendInformation", mf_3 → "SendAbord", mf_4 → "InformtheAgency", mf_5 → "BookTravel", mf_6 → "PayTravel", mf_7 → "ConfirmBooking", mf_8 → "OrderTicket"
target=	e_0 → "Requestforoffre", e_{12} → "OffreSP" e_1 → "Check?", e_2 → "checkTravelOffre", e_3 → "isthisofferinteresting?", e_4 → "informtheAgency?", e_5 → "Check?", e_{29} → "BookTravel", e_6 → "PayTravel", e_7 → "TicketReceived", e_8 → "BookingConfirmed", e_9 → "TransactionCompleted", e_{10} → "SendAbort", e_{11} → "TransactionAbort" e_{13} → "OtherOffer?", e_{14} → "MakeTravelOffer", e_{15} → "ExistMoreOffer", e_{16} → "SendInformation", e_{17} → "OfferCompleted", e_{18} → "OtherOffer", e_{19} → "Continue", e_{20} → "ExchangeSP", e_{21} → "BookingReceived", e_{22} → "PaymentReceived", e_{23} → "ConfirmBooking", e_{24} → "OrderTicket", e_{25} → "BookingCompleted", e_{26} → "OfferCompleted", e_{27} → "OfferAborded" mf_0 → "StartOfferManegement", mf_1 → "CheckTravelOffer", mf_2 → "ReceivedInformation", mf_3 → "ReceiveAbord", mf_4 → "StopSendingOffer", mf_5 → "BookingReceived", mf_6 → "PaymentReceived", mf_7 → "BookingConfirmed", mf_8 → "TicketReceived"
BoundaryEvent =	attachedTo("ReceivedInformation")= "CheckTravelOffer", isInterrupt("ReceivedInformation")= true attachedTo("StopSendingOffer") = "OfferSP", isInterrupt("StopSendingOffer") = true attachedTo("BoundaryEvent$_1$9y0yk9") = "ExchangeSP", isInterrupt("ReceiveAbord") = true
R(TravelAgency)=	{"StartOfferManagement","OfferSP","TimeOut","StopSending,Offer","Continue","ExchangeSP","ReceiveAbord","OfferAborted", "OfferCompleted", $e_{12}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}, e_{19}, e_{20}, e_{21}, e_{22}, e_{23}, e_{24}, e_{25}, e_{26}, e_{27}, e_{28}$}
R(Customer)=	{"StartTravelBooking","RequestforOffer","check?","CheckTravelOffer","istheofferinteresting?","Informtheagency","BookTravel", "PayTravel","BookingConfirmed","ReceivedInformation","SendAbort","TransactionAborted","TicketReceived","TransactionCompleted", $e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}$}

To get the content of a subprocess, we apply the R function on it. For example:

$$\begin{aligned} R(OfferSP) = {} & \{"StartOffers","OtherOffer?","ExistMoreoffer?","MakeTravel \\ & Offer","SendInformation","OffersCompleted"\} \\ R(ExchangeSP) = {} & \{"StartBooking","BookingReceived","PaymentReceived","ConfirmBooking", \\ & "OrderTicket","EndBooking"\} \end{aligned}$$

If we want to extract all the nodes within a process, we use the transitive closure function as follow:

$$\begin{aligned} R^+(TravelAgency) = {} & \{"StartOfferManagement","OfferSP","TimeOut","StopSending \\ & Offer","Continue","ExchangeSP","ReceiveAbord","OfferAborted", \\ & "OfferCompleted", e_{12}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}, e_{19}, e_{20}, e_{21}, e_{22}, e_{23}, e_{24}, \\ & e_{25}, e_{26}, e_{27}, e_{28},"StartOffers","OtherOffer?","ExistMoreoffer?", \\ & "MakeTravel,Offer","SendInformation","OffersCompleted","StartBookig", \\ & "BookingReceived","PaymentReceived","ConfirmBooking","Order \\ & Ticket","EndBooking"\} \end{aligned}$$

Example of a graph syntax. Using the example of Figure 2.8 again, the correspondence between the graphical notation of BPMN and the syntactic representation is exemplified in Table 4.1.

Auxiliary functions. For a graph $\widehat{G}$= (N, E, $\mathbb{M}$, cat_N, cat_E, $source$, $target$, R, msg_t, $attachedTo$, $isInterrupt$), we define the following auxiliary functions:

- $in : N \to 2^E$, returns the incoming edges of a node,

$$in(n) = \{e \in E \mid target(e) = n\}$$

- $out : N \to 2^E$, returns the outgoing edges of a node,

$$out(n) = \{e \in E \mid source(e) = n\}$$

- *procOf* : $N \to N^P$, returns the container process of a given node,

$$procOf(n) = \{p \mid \text{if and only if } n \in R^+(p)\}$$

2135 - *intype* : $N \times T_{Edges} \to 2^E$, returns a specified type of incoming edge for a given node

$$intype(n,t) = \{in \subseteq in(n) \mid \bigwedge_{e \in in} CatE(e) \in t \wedge t \in T_{Edges}\}$$

- *outtype* : $N \times T_{Edges} \to 2^E$, returns a specified type of outgoing edge for a given node

$$outtype(n,t) = \{out \subseteq out(n) \mid \bigwedge_{e \in out} CatE(e) \in t \wedge t \in T_{Edges}\}$$

It is important to enforce models to respect some well-formed rules before performing verification. We, therefore, define well-formed BPMN graphs as follows.

4.2.3 Well-formed BPMN graph.

A well-formed BPMN graph satisfies the following conditions. These rules extracted from the standard
2140 [3]:

- (C1) No incoming sequence flow edges for start events:

$$\forall n \in N,\ cat_N(n) \in SE \implies intype(n, SF) = \emptyset$$

- (C2) No outgoing sequence flow edges for end events:

$$\forall n \in N,\ cat_N(n) \in EE \implies outtype(n, SF) = \emptyset$$

- (C3) A sub-process contains exactly one None Start Event and no other start event types:

$$\begin{aligned}\forall n \in N^{SP},\ &|R(n) \cap \{nn \in N,\ cat_N(nn) = NSE\}| = 1\\ &\wedge R(n) \cap \{nn \in N,\ CatN(nn) \in \{MSE\}\} = \emptyset\end{aligned}$$

- (C4) A sub-process has a unique none end event node:

$$\forall n \in N^{SP}, |R(n) \cap \{nn \in N, cat_N(nn) = EE\}| = 1$$

- (C5) A sub-process node cannot contain a process node:

$$\forall n \in N, cat_N(n) \in SP \implies \forall nn \in R(n), cat_N(nn) \neq P$$

- (C6) For each process node, we require that:
  - it contains at least one initial node:

$$\forall n \in N, cat_N(n) = P \implies R(n) \cap \{nn \in N, cat_N(nn) = SE\} \neq \emptyset$$

  - it contains at least one end event node:

$$\forall n \in N, cat_N(n) = P \implies R(n) \cap \{nn \in N, cat_N(nn) = EE\} \neq \emptyset$$

- (C7) No looping edges: $\forall e \in E, source(e) \neq target(e)$
- (C8) No node isolation: $\forall n \notin N, (cat_N(n) = P) \implies (in(n) \neq \emptyset) \vee (out(n) \neq \emptyset)$
- (C9) A gateway that has a conditional edge must have a default edge:

$$\forall n \in N,\ (cat_N(n) \in G)\ \wedge\ (outtype(n, CSF) \neq \emptyset) \implies |outtype(n, DSF)| = 1$$

- (C10) No incoming message flow for send task, message end event, throw message intermediate event:
$$\forall n \in N,\ cat_N(n) \in \{ST, MEE, TMIE\} \implies intype(n, MF) = \emptyset$$

- (C11) No outgoing message flow for receive tasks, message start event, catch message intermediate event, boundary message intermediate event :
$$\forall n \in N,\ cat_N(n) \in \{RT, MSE, CMIE, BMIE\} \implies outtype(n, MF) = \emptyset$$

- (C12) A message flow edge connects two nodes of different processes:
$$\forall e \in E, cat_E(e) = MF \implies procOf(source(e)) \neq procOf(target(e))$$

- (C13) An event-based gateway have at least two outgoing edges:
$$\forall n \in N, cat_N(n) = EB \implies |out(n)| \geq 2$$

- (C14) Parallel and event-based gateways cannot have a conditional outgoing edge type:
$$\forall n \in N, cat_N(n) \in \{AND, EB\} \implies outtype(n, CSF) = \emptyset$$

- (C15) The outgoing edges of an inclusive or an exclusive gateway must be a combination between default sequence flows and conditional sequence flows, or all are of the normal sequence flow type:
$$\begin{aligned}\forall n \in N, cat_N(n) \in \{XOR, OR\} \implies & \forall e \in outtype(n, SF), cat_E(e) \in \{CSF, DSF\} \\ & \vee\ \forall e \in outtype(n, SF), cat_E(e) \in \{NSF\}\end{aligned}$$

- (C15) Elements that follow an event-based gateway can only be catching intermediate message 2145 events or receive tasks or timer intermediate catch events. Additionally, one cannot have both receive tasks and intermediate message events.

$$\begin{aligned}\forall n \in N cat_N(n) = EB \implies \quad & (\forall e \in outtype(n, SF), cat_N(target(e)) \in \{CMIE, RT, TICE\}) \\ & \wedge \begin{pmatrix} (\{e \in outtype(n, SF) \mid cat_N(target(e)) = RT\} = \emptyset) \\ \vee\ (\{e \in outtype(n, SF) \mid cat_N(target(e)) = CMIE\} = \emptyset) \end{pmatrix}\end{aligned}$$

- (C16) Message flows connect the throwing elements (send task, message end event, throw message intermediate event) with catching elements (receive task, message start event, catch message intermediate event, message boundary intermediate event):
$$\begin{aligned}\forall e \in E, cat_E(e) \in MF \implies & cat_N(source(e)) \in \{ST, MEE, TMIE\} \\ & \wedge\ cat_N(target(e)) \in \{RT, MSE, CMIE, MBE\}\end{aligned}$$

- (C17) Message catching elements must have at least one incoming message flow edge:
$$\forall n \in N, cat_N(n) \in \{RT, MSE, CMIE, MBE\} \implies |intype(n, MF)| \geq 1$$

- (C18) Message throwing elements must have at least one outgoing message flow edge:
$$\forall n \in N, cat_N(n) \in \{ST, MEE, TMIE\} \implies |outtype(n, MF)| \geq 1\}$$

- (C19) Receive task has at least one incoming message flow edge:
$$\forall n \in N, cat_N(n) \in RT \implies |intype(n, MF)| \geq 1$$

- (C20) Send task has at least one outgoing message flow edge:
$$\forall n \in N, cat_N(n) \in ST \implies |outtype(n, MF)| \geq 1$$

It should note that we do not require a specific structure of the BPMN graph for these rules. For example, we do not require these graphs to be well-balanced (when for each splitting gateway of a given type, there is a corresponding merging gateway of the same type). One can use an exclusive splitting 2150 gateway and merge its branches using a parallel gateway. Verification will be able to detect this is an erroneous model.

4.3 A Communication Model Representation

This thesis focuses on the BPMN collaboration diagrams where the communication models may present the backbones element in such a model. In general, the interactions between processes or any computing systems are built based on two categories: synchronous or asynchronous communication. In synchronous communications, the transmission of a piece of information - the message - is instantaneous (*i.e.*, the send and the receive of the data simultaneously). On the other hand, asynchronous communication splits the transmission into a send operation and a receive operation. In this Section, we consider classic communication models from the literature as well as a few variations. We integrate such models to the BPMN collaboration diagrams semantics to study the behaviour of such models in the presence of such control.

4.3.1 Communication Model

The interaction in the collaboration diagrams corresponds to a message passing between two processes. It corresponds to a couple of communication events: namely a *send* and a *receive*. An event or a set of communication events may occur on a process. However, each communication event can be a send event or a receive event associated with a message. Each event carries information about the type of the event (send, receive), the message, the process from where it occurs, and the process to where it will be fired. As the multi-instance characteristic is out of the scope in this thesis. We focus on *one-to-one* communication. In such communication, a given message is sent by a process, and it may be received by at most one process. Back to the example of Figure 2.8, it illustrates a point-to-point communication and the transmission of messages between the *TravelAgency* process and the *consumer* process. Formally, we define the peer to peer communication model using two predicates *send* and *receive* defined with *(from, to, message)* information as follow.

$$send/receive : \{p_1 \in N \mid cat_N(p_1) \in P\} \times \{p_2 \in N \mid cat_N(p_2) \in P\} \times \mathbb{M} \tag{4.1}$$

4.3.2 Communication Channel

Let *nets* be a set of communication channels. A channel is a label on messages. The content of messages is out of the scope here as we do not support data objects. A channel is not restricted to one sender and one receiver. Different processes can send messages on the same channel. Likewise, different processes can receive a message from the same channel. Yet, it is nonetheless a point-to-point communication abstraction because a given message still has exactly one sender and at most one receiver. If a single message is sent on a given channel and several peers expect to perform a reception from this channel, only one of them will be able to receive the message.

The channels can be global, local to a participant, associated with a pair of communicating processes, or local to each message. Thus, depending on the configuration of the channel, a great variety of communication models arises.

4.3.3 Generic Communication Models

We define seven communication models which differ in the order the messages can be sent or received. They are all the possible point-to-point models when considering local ordering (per process), causal ordering, and global ordering (absolute time) [179]. There are four variants of FIFO communication, Table 4.2 gives an overview of them. In addition, there are causal communication, *(Causal)*, pseudo-synchronous communication, *(RSC)*, and fully asynchronous communication *(Bag)*.

Table 4.2: *FIFO Communication Models Variants.*

FIFO pair	associates one sender with one receiver	1 → ▥ → 1
FIFO inbox	associates all the senders of a unique receiver	N ⇶ ▥ → 1
FIFO outbox	associates one sender with all its destinations	1 → ▥ ⇶ N
FIFO all	associates all the senders with all the receivers	N ⇶ ▥ ⇶ N

In the following, we define the structure of each model as a type called T_{net}, where

$$T_{net} \in \{bag, pair, inbox, outbox, causal, fifoall, RSC\}$$

The models are formally defined in Table 4.3 and are explained below. To simplify the notations, we include sequences of terms ($Seq[T]$) and bags ($Bag[T]$) as a part of the usual definition of ground terms in *first order logic*. Indeed, we use N^P to denote the subset of nodes of type P, *i.e.*, $N^P = \{n \in N \mid cat_N(n) \in P\}$. By abuse of notation, we may write N^P instead of $N^{\{P\}}$. We also assume some standard definitions and operations on terms:

- $\langle\rangle$ is the empty sequence
- *head*: $Seq[T] \to T$: returns the head of a sequence
- *tail*: $Seq[T] \to Seq[T]$: returns the tail of a sequence
- *append*: $Seq[T] \times T \to Seq[T]$: appends an element at the end of a sequence
- $\oplus$, $\ominus$: $Bag[T] \times Bag[T] \to Bag[T]$: union and difference of bags
- **Bag** is a multiset of messages. Formally:

$$bag \overset{def}{\equiv} Bag[N^P \times N^P \times \mathbb{M}]$$

  No order on message reception is imposed. Messages can overtake each other or be arbitrarily delayed. A bag or a set usually model it if messages are unique.

- **Fifo pair** is a queue of messages attached to each couple of processes. Formally:

$$pair \overset{def}{\equiv} N^P \times N^P \to Seq[\mathbb{M}]$$

  Messages between a couple of processes are received in their sending order. Messages from or to different processes are independently received. More precisely, if a process P_1 sends a message m_1 to process P_2, and later a message m_2 is sent to this same process, then m_2 cannot be received before m_1 (See Figure 4.2 and Figure 4.3 as example).

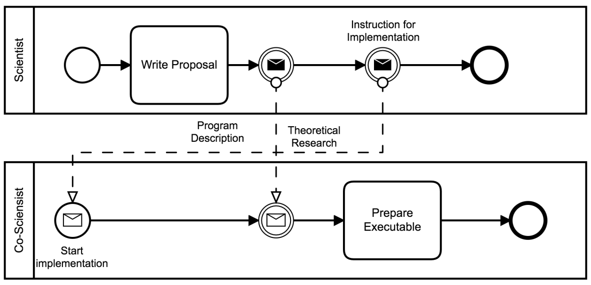


Figure 4.2: *A Non-FIFO Pair Execution. The sending of the theoretical research precedes the sending of the program description. Then program description is received before the theoretical research message, thus the model execution is not FIFO-Pair.*

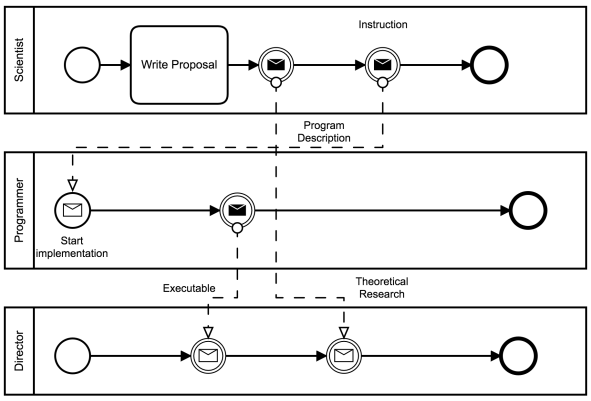


Figure 4.3: *A FIFO-Pair Execution. The sending of the theoretical research and the program description are received on different processes, so the execution is FIFO-Pair anyway.*

- **Fifo inbox** is an input queue attached to each process, where senders put messages. Formally:

$$inbox \overset{def}{\equiv} N^P \rightarrow Seq[N^P \times \mathbb{M}]$$

Each process has its own unique input queue, and senders add messages to this queue without blocking. Messages are consumed from this queue in their insertion order. This means that if a process P_1 sends a message m_1 to P_3, and later (but independently) a process P_2 sends a message
2210 m_2 to P_3, then m_2 cannot be received before m_1. This model is stricter than the Fifo pair as it requires a global order on the sending events (See Figure 4.5 and Figure 4.4 as an example).

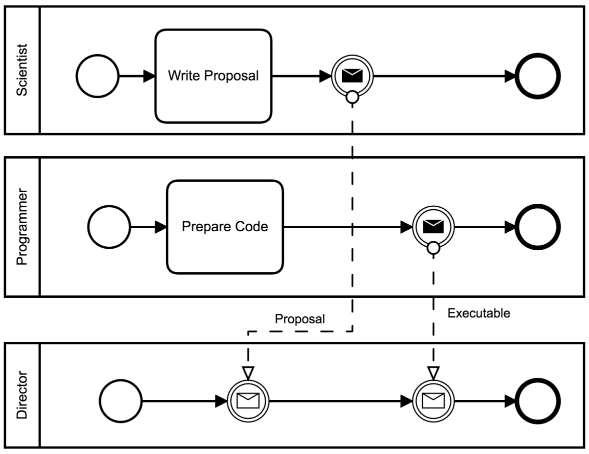


Figure 4.4: *A FIFO-Inbox Execution. The sending of proposal message precedes the executable, and the associated receptions that occur on the same process director happen in the same order.*

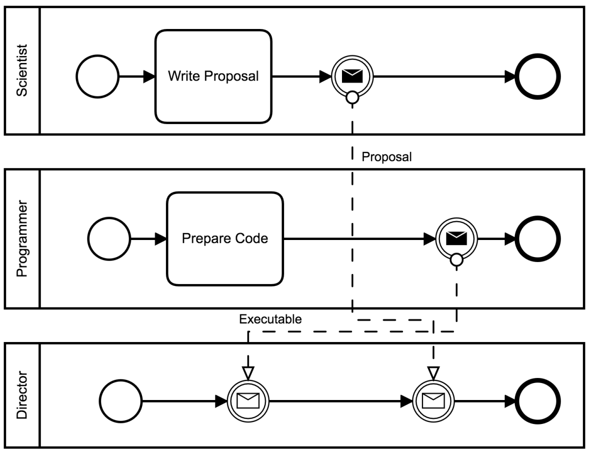


Figure 4.5: *A Non-FIFO Inbox Execution. The execution is not be FIFO inbox because the sending of proposal message precedes the executable, and the associated receptions that occur on the same process director happen in the reverse order.*

- **Fifo outbox** is an output queue attached to each process where messages are retrieved. Formally:

$$outbox \overset{def}{\equiv} N^P \rightarrow Seq[N^P \times \mathbb{M}]$$

Messages from the same process are received in their sending order. If a process P sends a message m_1 and later a message m_2 (to the same process or to another one), then m_2 cannot be received before m_1, even if the receptions occurs on distinct processes (See Figure 4.6 and Figure 4.7 as
2215 example).

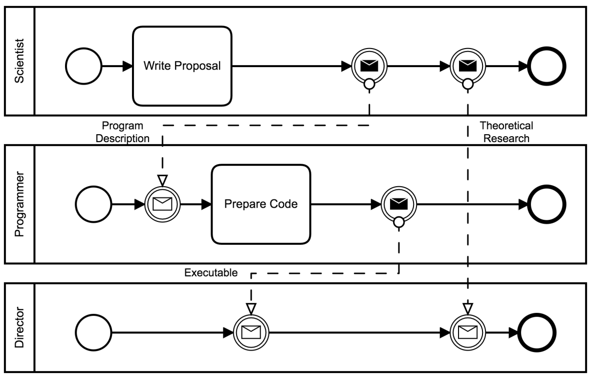


Figure 4.6: *A FIFO Outbox Execution. The sending of program description is sent before the theoretical research, both by Scientist process, and the associated receptions occur in this order.*

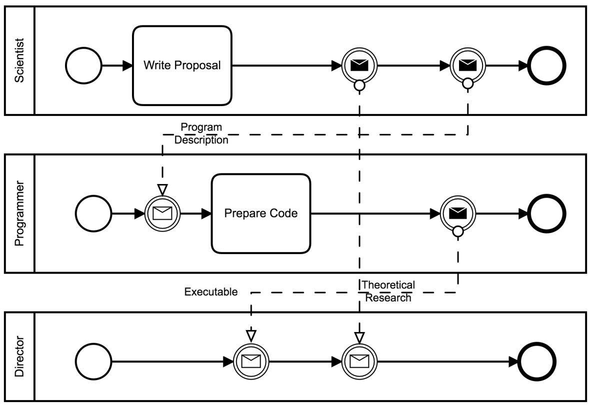


Figure 4.7: *A Non FIFO Outbox Execution. The execution is not be FIFO outbox because even the receptions occur on different processes, the receive event of program description occurred before the receive event of theoretical research.*

- **Fifo All** is a unique shared queue. Formally:

$$\mathit{fifoall} \overset{def}{\equiv} \mathit{Seq}[N^P \times N^P \times \mathbb{M}]$$

Messages are globally ordered, independently from their sender or receiver processes, and are received in the global sending order.

- **Causal.** Messages are received according to the causality of their sending [180]. Formally:

$$\begin{array}{l} VC \overset{def}{\equiv} [N^P \to \mathbb{N}] \text{ -- } \textit{a vector clock} \\ \mathit{causal} \in \mathit{Set}[N^P \times N^P \times \mathbb{M} \times VC] \times [N^P \to VC] \end{array}$$

If a message m_1 is causally sent before message m_2 (there exists a causal path from the sending of m_1 to the sending of m_2), then a process cannot receive m_2 before m_1. Figure 4.8 presents a model that deadlocks with causal communication. In this example, a scientist writes a proposal, sends the proposal to the client, and sends its description to its financial department. Based on the description, the financial department computes a quote and sends it to the client. Without causal communication, the client can receive the quote before the proposal; with causal communication, the quote cannot be delivered because it causally depends on the proposal that must be received first. A usual implementation of this model uses causal histories [181, 182] or vector/matrix logical clocks [183] as presented here[1]. The state of the network *mnet* holds a couple: the set of messages in transit (with their associated vector clock) and the vector clocks of each process.

[1]A vector clock is a logical clock that tracks the causal dependencies between send and receive events. They are not to be confused with the clocks used for timing constraints in Chapter 5.

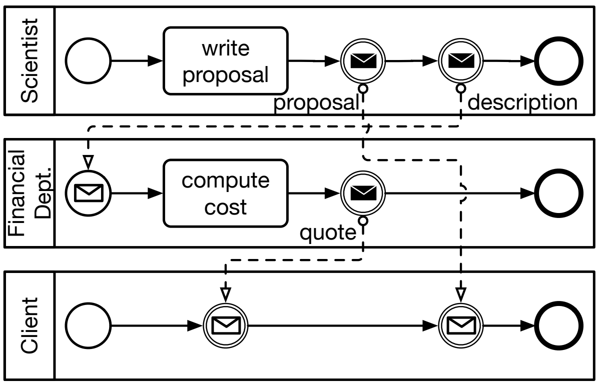


Figure 4.8: *A Non-Causal Execution. The sending of the proposal causally precedes the sending of the description which causally precedes the sending of the quote. Then the quote cannot be consumed before the proposal on the client in the causal communication model.*

- **RSC (Realisable with Synchronous Communication)** is a shared, 1-sized, buffer for all processes. Formally:

$$net_{RSC} \overset{def}{\equiv} (N^P \times N^P \times \mathbb{M})$$

Send and receive events are strictly alternate. If the couple (send event, reception event) is considered atomic, this corresponds to synchronous communication [184].

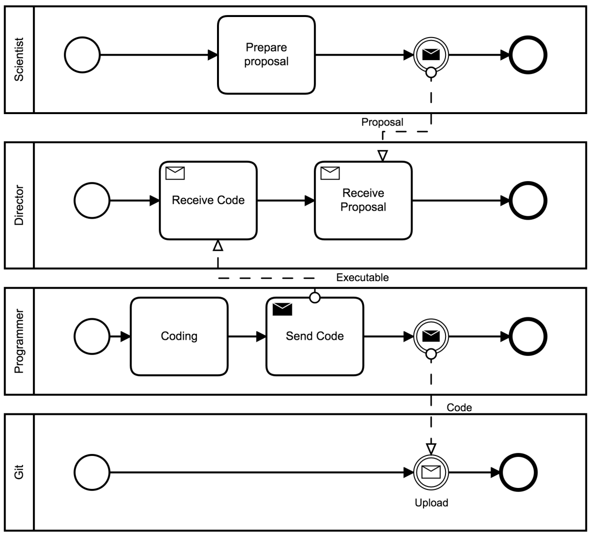


Figure 4.9: *A Non-RSC Execution. The sending of the code message is occurred precedes the reception of the proposal message. Two messages must not be in transit at the same time.*

4.4 A FOL Semantics for BPMN Collaborations

To maintain traceability with the standard, we use a token-based approach to define the semantics. We formalise the way the token transit between edges and nodes that compose the collaboration diagram.

Generally, to apply model-checking techniques, the system is represented by a transition system [185]. It is a model that defines the states and actions that provoke transitions between these states. Mainly, there are two ways to model a system of transitions using operational or declarative languages. Transitions are expressed using assignment instructions in the operational language, either with imperative, functional language control flow (*e.g.*, Promela, the language of the Spin model-checker) or by using variants of Diskstra's guard commands (*e.g.*, Murphi [186] or SMV [187]). Second, transitions are expressed with constraints in the declarative language, either on the complete execution or, more often, on individual steps. The idea of this idiom takes root in the first works on program verification like the specification

Table 4.3: *Encoding of the Communication Models in First-Order Logic.*

Network model	Definition
Bag	$initnet \overset{def}{\equiv} \emptyset$
	$send(from, to, m) \overset{def}{\equiv} mnet' = mnet \oplus \{\langle from, to, m\rangle\}$
	$receive(from, to, m) \overset{def}{\equiv} \langle from, to, m\rangle \in mnet$
	$\quad \wedge\ mnet' = mnet \ominus \{\langle from, to, m\rangle\}$
Fifo Pair	$initnet \overset{def}{\equiv} [p, q \in N^P \mapsto \langle\rangle]$
	$send(from, to, m) \overset{def}{\equiv} mnet'(from, to) = append(mnet(from, to), m)$
	$receive(from, to, m) \overset{def}{\equiv} m = head(mnet(from, to))$
	$\quad \wedge\ mnet'(from, to) = tail(mnet(from, to))$
Fifo Inbox	$initnet \overset{def}{\equiv} [p \in N^P \mapsto \langle\rangle]$
	$send(from, to, m) \overset{def}{\equiv} mnet'(to) = append(mnet(to), \langle from, m\rangle)$
	$receive(from, to, m) \overset{def}{\equiv} \langle from, m\rangle = head(mnet(to))$
	$\quad \wedge\ mnet'(to) = tail(mnet(to))$
Fifo Outbox	$initnet \overset{def}{\equiv} [p \in N^P \mapsto \langle\rangle]$
	$send(from, to, m) \overset{def}{\equiv} mnet'(from) = append(mnet(from), \langle to, m\rangle)$
	$receive(from, to, m) \overset{def}{\equiv} \langle to, m\rangle = head(mnet(from))$
	$\quad \wedge\ mnet'(from) = tail(mnet(from))$
Fifo All	$initnet \overset{def}{\equiv} \langle\rangle$
	$send(from, to, m) \overset{def}{\equiv} mnet' = append(mnet, \langle from, to, m\rangle)$
	$receive(from, to, m) \overset{def}{\equiv} \langle from, to, m\rangle = head(mnet) \wedge mnet' = tail(mnet)$
Causal	$initnet \overset{def}{\equiv} \emptyset \times [p \in N^P \mapsto [q \in N^P \mapsto 0]]$
	$send(from, to, m) \overset{def}{\equiv}$
	$\quad mnet'[1] = mnet[1] \cup \{\langle from, to, m, vc[from]\rangle\}$
	$\quad \wedge\ mnet'[2] = vc$
	with $vc \overset{def}{\equiv} [p \in N^P \rightarrow [q \in N^P \rightarrow$
	$\qquad$ if $p = from \wedge q = from$ then $mnet[2][p][q] + 1$ else $mnet[2][p][q]$ $]]$
	$receive(from, to, m) \overset{def}{\equiv}$
	$\quad \exists msg \in mnet[1], msg[1] = from \wedge msg[2] = to \wedge msg[3] = m$
	$\qquad \wedge\ \neg(\exists msg_2, msg_2 \neq msg \wedge msg_2[2] = to \wedge\ \forall p \in N^P, msg_2[4][p] \leq msg[4][p])$
	$\qquad \wedge\ mnet'[1] = mnet[1] \setminus \{msg\}$
	$\qquad \wedge\ mnet'[2] = [p \in N^P \mapsto$ if $p = to$ then $Sup(mnet[2][p], m[4])$ else $mnet[2][p]]$
RSC	$initnet \overset{def}{\equiv} \emptyset$
	$send(from, to, m) \overset{def}{\equiv} mnet = \emptyset \wedge mnet' = \{\langle from, to, m\rangle\}$
	$receive(from, to, m) \overset{def}{\equiv} \langle from, to, m\rangle \in mnet \wedge mnet' = \emptyset$

of the declarative languages VDM [188], Larch [189], or Z [190], which are essentially the pre and post conditions of Hoare triplets [39].

In this section, we define the semantics of our formalism using the First Order Logic (FOL) and the declarative idiom. The execution semantics of BPMN is defined based on the transition notion. A transition is enabled first before being fired for moving from a state to another state. Thus, the enabling notion corresponds to a precondition, while the firing notion corresponds to a postcondition. In our case, a transition is said to be enabled when some preconditions are met (to allow the firing of the transition). We distinguish two cases: (i) a node is ready to start its execution, (ii) a node is ready to complete its execution. However, a transition is said to be fired when some postconditions are met (determines the node activation or completing). Thus, we distinguish two cases: (iii) firing a transition on a node that can start; (iv) firing a transition on a node that can complete. To simplify our model, we merge (i) and (iii) to define the starting action and (ii) and (iv) to determine the completing action. Thus, we describe the movement of tokens based on the node types and the two predicates, starting predicate St and completing predicate Ct defined for each node type. We note that some nodes only have a start transition (*e.g.*, end events), and others only have a completion transition (*e.g.*, gateways). When a node defines only one of the two predicates, the other one is considered false.

The semantics (Section 4.4.0.2) relies on a notion of *state* of the BPMN graph (Section 4.4.0.1) to define the St and Ct predicates. Further, the semantics is parameterised by a type T_{net} that encapsulates the properties of the communication network using an initialisation function, *initnet*, and two predicates, *send* and *receive*.

4.4.0.1 State

A state of a BPMN graph gives a marking for the nodes and the edges, together with a state for the communication network.

Definition 4.4.1 (State)**.** The state of a BPMN graph is a tuple $s = (mn, me, mnet)$ such that:

- $mn : N \to \mathbb{N}$, is a function assigning a natural number marking to each node.
- $me : E \to \mathbb{N}$, is a function assigning a natural number marking to each edge.
- $mnet : T_{net}$, is the state of the communication network.

States denotes the set of all states of a BPMN graph.

Definition 4.4.2 (Initial state)**.** The initial state of a BPMN graph, denoted by $s_o = (mn_0, me_0, mnet_0)$, is such that:

- the start nodes of the processes hold a token, all other nodes are unmarked:

$$\forall n \in N, mn_0(n) = \begin{cases} 1 & if\ cat_N(n) \in SE \land (\exists p \in N, cat_N(p) = P \mid n \in R(p)) \\ 0 & otherwise. \end{cases}$$

- all edges are unmarked: $\forall e \in E, me_0(e) = 0$
- the network is empty: $mnet_0 = initnet$

4.4.0.2 Semantics

Here, we define the execution semantics of BPMN based on those mentioned above, St and Ct predicates for each type of node in the BPMN graph and based on its notion of state. In the semantics, let $s = (mn, me, mnet)$ and $s' = (mn', me', mnet')$ denote two states. Additionally, we consider the predicate $\triangle$ that denotes marking equality but for nodes and edges given as parameters. Hence, $\triangle(X)$ means "*nothing changes except for X*":

$$\triangle(X) \overset{def}{\equiv} \forall n \in N \setminus X, mn'(n) = mn(n) \land \forall e \in E \setminus X, me'(e) = me(e)$$

Similarly, Ξ denotes that the state of the network does not change: $\Xi \overset{def}{\equiv} mnet' = mnet$.

2275 **Start nodes.** There are two starting node types for the instantiation of the process: the none start event (NSE) and the message start event (MSE).

The behaviour of an NSE is defined only by a completing predicate. It consumes its token and generates one token on all of its outgoing sequence flow edges. If it is the initial node of a process p, it activates p by generating a token on it. When an NSE is defined within a sub-process p, its activation is conditioned by the activation of p. Formally:

$$\begin{aligned}\forall n \in N, Ct(n) \overset{def}{\equiv}\ & cat_N(n) = NSE \wedge (mn(n) >= 1) \wedge (mn'(n) = mn(n) - 1)\\ & \wedge \forall e \in outtype(n, SF), (me'(e) = me(e) + 1)\\ & \wedge \left(\begin{array}{l} \left(\begin{array}{c} \exists p \in N, p = R^{-1}(n) \wedge (cat_N(p) = P) \\ \wedge (mn(p) = 0) \wedge \ (mn'(p) = 1) \\ \wedge \triangle (\{n, p\} \cup outtype(n, SF) \wedge \Xi) \end{array} \right) \\ \vee \left(\begin{array}{l} \exists sp \in N, sp = R^{-1}(n) \wedge (cat_N(sp) = SP) \\ \wedge \triangle (\{n\} \cup outtype(n, SF)) \wedge \Xi \end{array} \right) \end{array} \right)\end{aligned}$$

Example. Figure 4.10 presents a process p with a none start event (nse), an abstract task ($task$), and a none end event (nee). It shows a token within nse node, which is represented by a green token. nse completes by consuming this token and produces one on P and its outgoing sequence flow ($e1$).

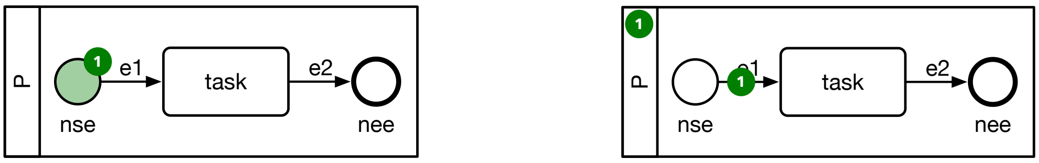


Figure 4.10: *Completing Behaviour of a None Start Event. Before (left) and after (right) application of the Ct rule.*

The behaviour of a message start event (MSE) is defined by a completing predicate. An MSE is enabled if it has a token and there is a message offer on one of its incoming sequence flow edges. It completes by consuming the message, generating one token on all of its outgoing sequence flow edges, and activating the process p by generating a token on it. Formally:

$$\begin{aligned}\forall n \in N, Ct(n) \overset{def}{\equiv}\ & cat_N(n) = MSE \wedge (mn(n) = 1) \wedge (mn'(n) = mn(n) - 1)\\ & \wedge \forall e \in outtype(n, SF), (me'(e) = me(e) + 1)\\ & \wedge \exists em \in intype(n, MF), (me(em) \geq 1) \wedge (me'(em) = me(em) - 1)\\ & \quad \wedge receive(procOf(source(em)), procOf(n), msg_t(em))\\ & \quad \wedge \exists p \in N, cat_N(p) = P \wedge n \in R(p) \wedge (mn(p) = 0) \wedge (mn'(p) = 1)\\ & \qquad \wedge \triangle (\{n, p, em\} \cup outtype(n, SF))\end{aligned}$$

2280 **Example.** Consider the example of Figure 4.10 again. By replacing the none start event with a message start event (mse), we get the model in Figure 4.11. It represents the complete execution semantics behaviour. The left-hand-side of Figure 4.11 shows that there is a token on the start node and a message offer ($m1$) on the incoming message flow edge of mse. This latter completes by consuming the message according to the chosen communication model and producing a token on the process and on all its
2285 outgoing edges.

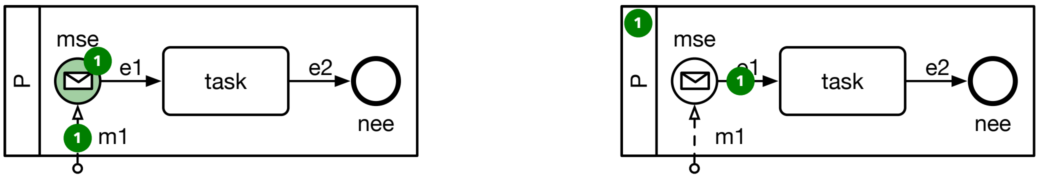


Figure 4.11: *Completing Behaviour of a Message Start Event. Before (left) and after (right) application of the Ct rule.*

Ending nodes We have three ending node types for the termination of a process: none end event (NEE), terminate end event (TEE), and message end event (MEE).

The behaviour of an NEE node is defined only by a starting predicate: it is enabled if it has at least one token on one of its incoming edges. It starts by consuming this token and adding one to itself. Formally:

$$\forall n \in N, St(n) \overset{def}{\equiv} cat_N(N) = NEE \wedge \exists e \in intype(n, SF), (me(e) \geq 1) \wedge (me'(e) = me(e) - 1) \\ \wedge\ (mn'(n) = mn(n) + 1) \wedge \triangle(\{n, e\}) \wedge Xi$$

Example. Figure 4.12 presents the execution semantics of an end node (nee). The left-hand side of the figure shows the enabling of the nee by the presence of a token on its incoming edge ($e2$). The right-hand side of the figure shows the starting behaviour. It consumes the token from $e2$ and generates a token on it. 2290

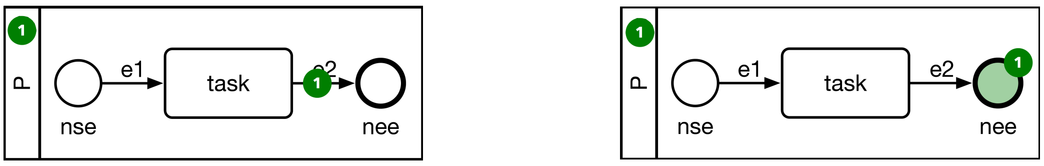


Figure 4.12: *Starting Behaviour of None End Event. Before (left) and after (right) application of the St rule.*

A TEE node is defined only by a starting predicate: it is enabled if it has at least one token on one of its incoming edges. It behaves like a none end event by consuming a token from one of its incoming sequence flows and generates a token on itself. Besides, it does the additional work of dropping down all the remaining tokens of the process or sub-processes to which it belongs. Formally:

$$\forall n \in N, St(n) \overset{def}{\equiv} cat_N(n) = TEE \\ \wedge\ \exists e \in intype(n, SF), (me(e) \geq 1) \wedge (mn'(n) = 1) \\ \wedge\ \exists p \in N, cat_N(p) \in \{P, SP\}, (n \in R(p)) \\ \wedge\ \forall nn \in ((R^+(p) \cap N) \setminus \{n\}), (mn'(nn) = 0) \\ \wedge\ \forall ee \in (R^+(p) \cap E), (me'(ee) = 0) \\ \wedge\ \triangle\ (R^+(p)) \wedge \Xi$$

Example. The left-hand side of Figure 4.13 shows the enabling of a terminate end node (tee) by the presence of a token on its incoming edge ($e7$). The right-hand side of the figure shows the starting execution of the node. It consumes the token from $e7$ and generates a token on itself. Besides, it affects all the existing executions in parallel (here $e3$ and $e4$) by dropping down all their tokens. 2295

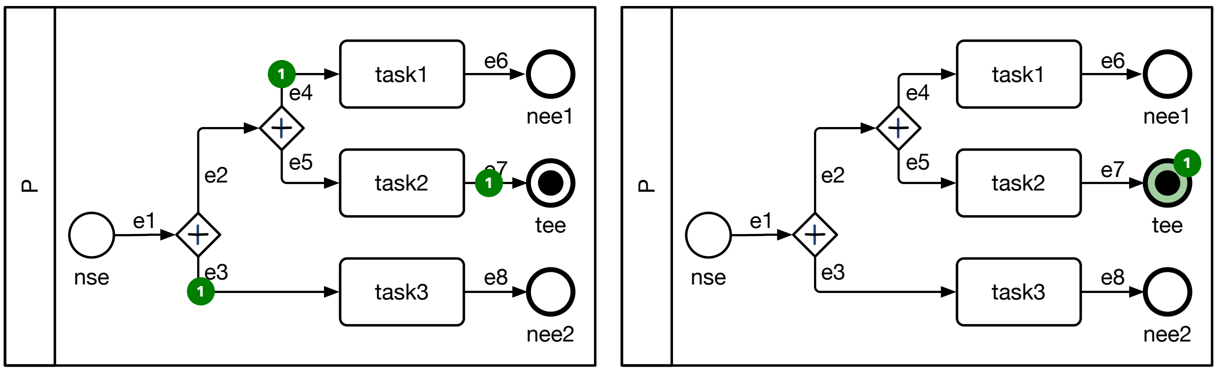


Figure 4.13: *Starting Behaviour of a Terminate End Event. Before (left) and after (right) application of the St rule.*

An MEE node is defined only by a starting predicate. It is enabled to start if it has a token on one of its incoming edges. Then, it moves the token from one of its incoming edges to itself and sends a message on the network according to the configured communication model. Formally:

$$\begin{aligned}\forall n \in N, St(n) \overset{def}{\equiv}\ & cat_N(n) = MEE \\ & \wedge\ (mn'(n) = mn(n) + 1) \\ & \wedge\ \exists e \in intype(n, SF), (me(e) \geq 1) \wedge (me'(e) = me(e) - 1) \\ & \wedge\ \exists ee \in outtype(n, MF), (me'(ee) = me(ee) + 1) \\ & \qquad \wedge\ send(procOf(n), procOf(target(ee)), msg_t(ee)) \\ & \qquad \wedge \triangle (\{n, e, ee\})\end{aligned}$$

Example. Figure 4.14 shows the starting behaviour of the message end event (mee). It starts by 2300 consuming the token from the incoming edge ($e2$), producing a token on itself, and sending a message $m1$ on the network.

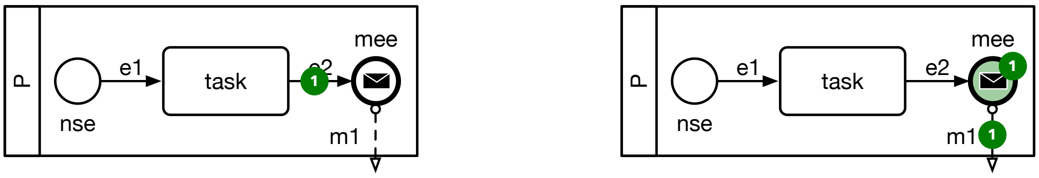


Figure 4.14: *Starting Behaviour of a Message End Event. Before (left) and after (right) application of the St rule.*

Activity nodes Two kinds of activity nodes have to be taken into account: the abstract tasks (AT) and the subprocesses (SP).

A starting and completing predicates define the behaviour of an abstract task node (AT). It is enabled 2305 to start if at least one token is present on one of its incoming edges and it does not already own a token. Then, it starts by consuming a token from one of its incoming edges and produces one on itself. An AT node is completed by consuming one token from itself and adding one token on each of its outgoing edges. Note that an interrupting boundary event may end an abstract task (see MBE, pages 87). Formally:

$$\begin{aligned}\forall n \in N, St(n) \overset{def}{\equiv}\ & cat_N(n) = AT \\ & \wedge\ \exists e \in intype(n, SF), (me(e) \geq 1) \wedge (me'(e) = me(e) - 1) \\ & \quad \wedge\ (mn(n) = 0) \wedge (mn'(n) = mn(n) + 1) \\ & \quad \wedge \triangle (\{n, e\}) \wedge \Xi\end{aligned}$$

$$\begin{aligned}\forall n \in N, Ct(n) \overset{def}{\equiv}\ & cat_N(n) = AT \wedge (mn(n) = 1) \wedge (mn'(n) = mn(n) - 1) \\ & \wedge\ \forall e \in outtype(n, SF), (me'(e) = me(e) + 1) \\ & \wedge \triangle (\{n\} \cup outtype(n, SF)) \wedge \Xi\end{aligned}$$

Example. Figure 4.15 shows the starting behaviour of the abstract task *task*. It starts by consuming 2310 the token from $e1$ and generating a token on itself.

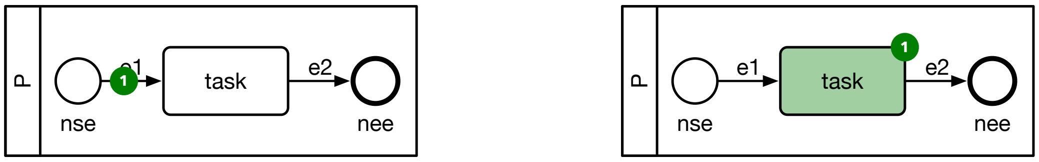


Figure 4.15: *Starting Behaviour of an Abstract Task activity. Before (left) and after (right) application of the St rule.*

Figure 4.16 shows its completing behaviour. It consumes the token from itself and generates one on its outgoing edge $e2$.

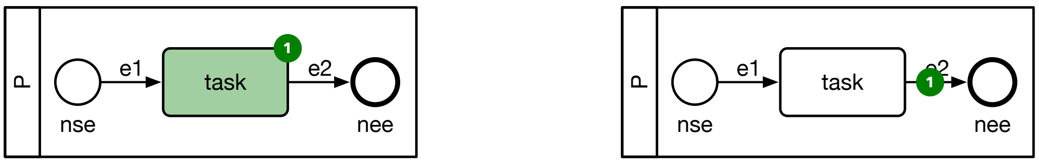


Figure 4.16: *Completing Behaviour of an Abstract Task Activity. Before (left) and after (right) application of the Ct rule.*

The behaviour of a subprocess activity SP node extends the one of an AT node with some additional conditions: when it is enabled, a sub-process adds a token to the start event it contains. It completes when at least one end event it contains has some tokens, and none of its edges or nodes is still active (*i.e.*, owning a token). Note that, like an abstract task, a subprocess may also be ended by an interrupting boundary event (see MBE, pages 87). Formally:

$$\begin{aligned}\forall n \in N, St(n) \overset{def}{\equiv}\ & (cat_N(n) = SP) \wedge \exists e \in intype(n, SF), (me(e) \geq 1) \wedge (me'(e) = me(e) - 1)\\ & \wedge\ (mn(n) = 0) \wedge (mn'(n) = mn(n) + 1)\\ & \wedge\ \forall ns \in (N^{NSE} \cap R(n)), (mn'(ns) = mn(ns) + 1)\\ & \wedge\ \triangle\ (\{e, n\} \cup (\{nse \in N, cat_N(nse) = NSE\} \cap R(n))) \wedge \Xi\end{aligned}$$

$$\begin{aligned}\forall n \in N, Ct(n) \overset{def}{\equiv}\ & (cat_N(n) = SP) \wedge (mn(n) = 1) \wedge (mn'(n) = mn(n) - 1)\\ & \wedge\ \forall e \in R(n) \cap E, (me(e) = 0)\\ & \wedge\ \exists n_{ee} \in R(n) \wedge cat_N(n_{ee}) \in EE) \wedge (mn(n_{ee}) \geq 1)\\ & \wedge\ \forall nn \in R(n) \cap N, (mn(nn) \geq 1 \Rightarrow cat_N(nn) \in EE)\\ & \wedge\ \forall nn \in (R(n) \cap cat_N(nn) \in EE), (mn'(nn) = 0)\\ & \wedge\ \forall e \in outtype(n, SF), (me'(e) = me(e) + 1)\\ & \wedge\ \triangle\ (\{n\} \cup (R(n) \cap \{nee \in N, cat_N(nee) \in EE\}) \cup outtype(n, SF)) \wedge \Xi\end{aligned}$$

Example. Figure 4.17 shows the starting behaviour of the subprocess activity (SP). It starts by consuming a token from its incoming edge ($e1$) and generating a token on itself and on its start event
2315 ($nse1$).

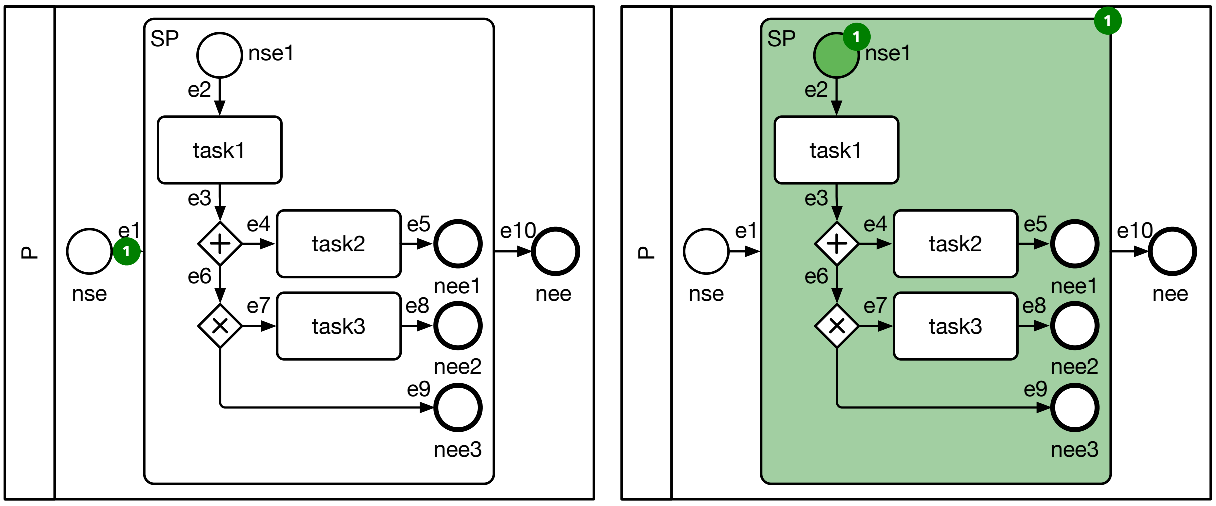


Figure 4.17: *Starting Behaviour of a subprocess Activity. Before (left) and after (right) application of the St rule.*

Figure 4.18 shows that even if there is a token on one of the end events, here $nee3$, the subprocess can not execute its completing transition: to complete, SP must wait until the token on $e5$ has given place to one on $nee1$.

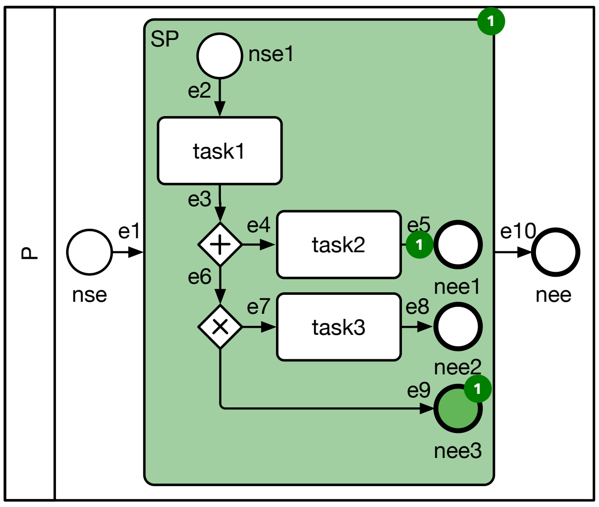


Figure 4.18: *A Sub-Process Activity not Ready to Complete: a token is still present on one of its edges.*

Communication The semantics for MSE and MEE have been presented above. The remaining communicating elements, $TMIE$, $CMIE$, MBE, ST and RT require additional conditions for starting and completing due to the presence of sending/reception predicates. All these elements require a token on one of their incoming edges to be enabled.

The ST and RT are enabled when they have a token on their incoming edges and no token on them. Then, they start executing by moving this token inside. Finally, ST completes by sending a message on all its outgoing edges regarding the chosen communication model (which affects the network state) and producing a token on all its outgoing edges.

A ST is not necessarily instantaneous as a send may block, for instance, with synchronous communication, RSC or a bounded size network. Another important point is that a boundary event, such as a timeout, can be attached to a receive or send task and not to an event. For all these reasons, we have chosen to make these tasks non-atomic. This allows distinguishing a send task from a ThrowMessageIntermediateEvent, and a receive task from a CatchMessageIntermediateEvent. Note that the semantics in the standard is ambiguous since there are two contradictory aspects: tasks in BPMN are non-atomic while the purpose of a send task is only to send a message, which is intrinsically atomic. Formally:

$$\begin{aligned}\forall n \in N, St(n) \overset{def}{\equiv}\ & cat_N(n) = ST \wedge \exists e \in intype(n, T_{SP}), (me(e) \geq 1) \wedge (me'(e) = me(e) - 1)\\ & \wedge (mn(n) = 0) \wedge (mn'(n) = mn(n) + 1)\\ & \wedge \triangle(\{n, e\}) \wedge \Xi\end{aligned}$$

$$\begin{aligned}\forall n \in N, Ct(n) \overset{def}{\equiv}\ & cat_N(n) = ST \wedge (mn(n) = 1) \wedge (mn'(n) = mn(n) - 1)\\ & \wedge \forall e \in outtype(n, SF), (me'(e) = me(e) + 1)\\ & \wedge \exists ee \in outtype(n, MF), (me'(ee) = me(ee) + 1)\\ & \quad \wedge send(procOf(n), procOf(target(ee)), msg_t(ee))\\ & \quad \wedge \triangle(\{n, ee\} \cup outtype(n, SF))\end{aligned}$$

Example. Figure 4.19 presents the starting behaviour of a send task activity ($task$). It shows the same starting behaviour as the one presented in Figure 4.15.

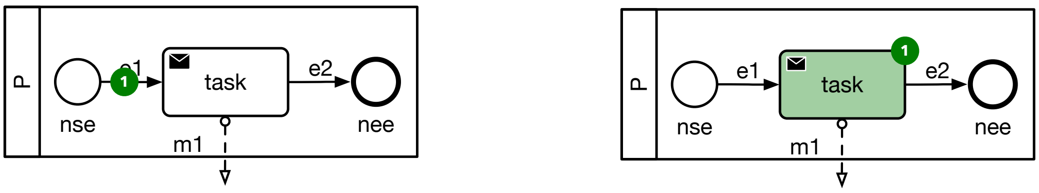


Figure 4.19: *Starting Behaviour of a Send Task Activity. Before (left) and after (right) application of the St rule.*

Figure 4.20 shows the completing behaviour, $task$ completes by consuming one token from it and by generating a token on its outgoing edge $e2$ and producing a message $m1$ on the network according to the chosen communication model.

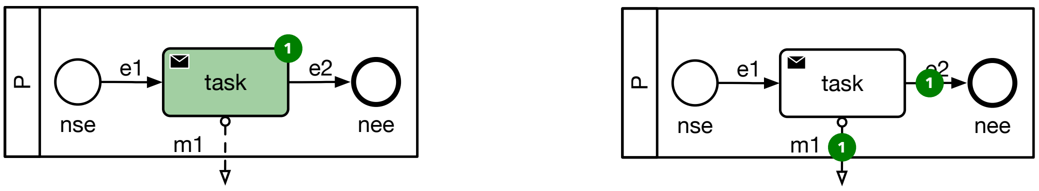


Figure 4.20: *Completing Behaviour of a Send Task Activity. Before (left) and after (right) application of the Ct rule.*

A receive task (RT) has a complementary behaviour to the send task (ST). It is enabled to complete only if it is active (*i.e.*, it has a token) and it has a message on one of its incoming message flows. RT completes by consuming the message offer, updating the network state, and producing a token on all its outgoing edges. Formally:

$$\begin{aligned}\forall n \in N, St(n) \overset{def}{\equiv}\ & cat_N(n) = RT \wedge \exists e \in intype(n, SF), (me(e) \geq 1) \wedge (me'(e) = me(e) - 1)\\ & \wedge (mn(n) = 0) \wedge (mn'(n) = mn(n) + 1)\\ & \wedge \triangle(\{n, e\}) \wedge \Xi\end{aligned}$$

$$\begin{aligned}\forall n \in N, Ct(n) \overset{def}{\equiv}\ & cat_N(n) = RT \wedge (mn(n) = 1) \wedge (mn'(n) = mn(n) - 1)\\ & \wedge \forall e \in outtype(n, SF), (me'(e) = me(e) + 1)\\ & \wedge \exists ee \in intype(n, MF), (me(ee) \geq 1) \wedge (me'(ee) = me(ee) - 1)\\ & \quad \wedge receive(procOf(source(ee)), procOf(n), msg_t(ee))\\ & \quad \wedge \triangle(\{n, ee\} \cup outtype(n, SF))\end{aligned}$$

Example. The starting of the receive task activity is similar to the one presented for the abstract task in Figure 4.15. Figure 4.21 shows that the receive task activity ($task$) can complete if it has a token on itself and a message $m1$ on its incoming message flow. It completes by consuming its token, receiving the message from the network, and producing a token on its outgoing edge ($e2$).

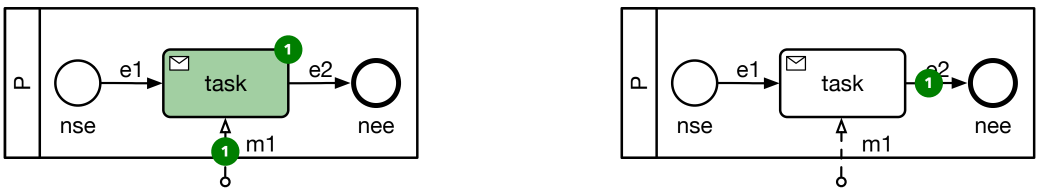


Figure 4.21: *Completing Behaviour of a Receive Task Activity. Before (left) and after (right) application of the Ct rule.*

A throw message intermediate event ($TMIE$) defines only the starting behaviour. It is enabled to start if it has a token on one of its incoming edges. It starts by consuming the token from this incoming edge, sending a message on the network according to the chosen communication model, and producing a token on all its outgoing edges. Formally:

$$\begin{aligned}\forall n \in N, St(n) \overset{def}{\equiv}\ & cat_N(n) = TMIE\\ & \wedge \exists ein \in intype(n, SF), (me(ein) \geq 1) \wedge (me'(ein) = me(ein) - 1)\\ & \wedge \forall e \in outtype(n, SF), (me'(e) = me(e) + 1)\\ & \wedge \exists eout \in outtype(n, MF), (me'(eout) = me(eout) + 1)\\ & \quad \wedge send(procOf(n), procOf(target(eout)), msg_t(eout))\\ & \quad \wedge \triangle(\{ein, eout\} \cup outtype(n, SF))\end{aligned}$$

Example. Figure 4.22 shows the starting behaviour of a throw message intermediate event (*Send Notif.*). It starts by consuming the token from its incoming edge ($e1$), producing a token on its outgoing edge ($e2$), and sending a message $m1$ on the network.

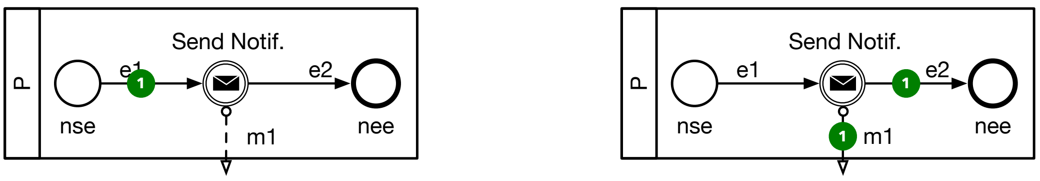


Figure 4.22: *Starting Behaviour of a Throw Message Intermediate Event. Before (left) and after (right) application of the St rule.*

A catching message intermediate event ($CMIE$) is an instantaneous event with only a starting transition. It is enabled if a message offer is on one of its incoming message flow edges and a token on one of its incoming sequential flow edges. It starts by consuming the token from this incoming edge, receiving the message from the incoming message flow according to choose the communication model, and producing a token on all its outgoing edges. Formally:

$$\begin{aligned}\forall n \in N, St(n) \overset{def}{\equiv}\ & cat_N(N) = CMIE \\ & \wedge \exists e_1 \in intype(n, SF), (me(e_1) \geq 1) \wedge (me'(e_1) = me(e_1) - 1) \\ & \wedge \forall e_2 \in outtype(n, SF), (me'(e_2) = me(e_2) + 1) \\ & \wedge \exists ein \in intype(n, MF), (me(ein) \geq 1) \wedge (me'(ein) = me(ein) - 1) \\ & \quad \wedge receive(procOf(source(ein)), procOf(n), msg_t(ein)) \\ & \quad \wedge \triangle\, (\{e_1, ein\} \cup outtype(n, SF))\end{aligned}$$

Example. Figure 4.23 shows the starting behaviour of a catching message intermediate event (*Receive Notif.*). It starts by consuming the token from $e1$, receiving the message $m1$ from the medium, and producing a token on its outgoing edge, $e2$.

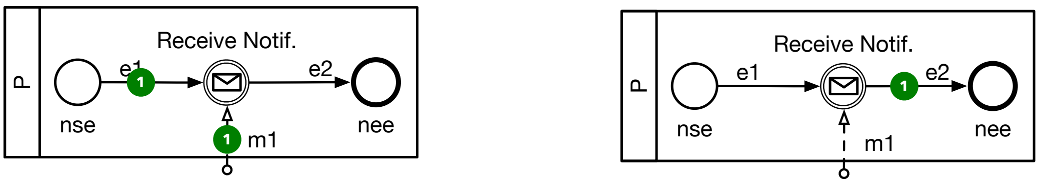


Figure 4.23: *Starting Behaviour of a Catching Message Intermediate Event. Before (left) and after (right) application of the St rule.*

Boundary Events A message boundary event (MBE) defines only the starting behaviour. An MBE is ready to start if it has a message offer on one of its incoming message flows and if the activity on which it is attached has a token. An MBE may have either an interrupting behaviour or a non-interrupting one. In the latter case, the MBE starts by receiving a message and generating a token on all its outgoing edges. For an interrupting behaviour, the MBE also starts by cancelling the activity to which it is attached, which is possible only if this activity is not a sub-process in a completing step. This is checked using the *mayComplete* predicate that is formally defined below. Cancelling an activity involves dropping all its tokens. After that, the MBE produces a token on each of its outgoing edges. Formally:

Auxiliary functions To formalise the semantics of a message boundary event, we define an auxiliary function.

- $mayComplete(n) : \{nn \in N, cat_N(nn) = SP\} \rightarrow Bool$, returns true if the subprocess may complete, *i.e.*, if there are no tokens on its elements except for its end event nodes where there is at least one that holds some tokens.

$$\begin{aligned}\forall n \in N, mayComplete(n) \overset{def}{\equiv}\ & cat_N(n) = SP \wedge (mn(n) \geq 1) \\ & \wedge \forall e \in (R(n) \cap E), (me(e) = 0) \\ & \wedge \exists nn \in R(n), (cat_N(nn) \in EE) \wedge (mn(nn) \geq 1) \\ & \wedge \forall x \in R(n), ((cat_N(x) \in EE) \vee (mn(x) = 0))\end{aligned}$$

$$
\begin{array}{lll}
St_{interrupting}(n, act, ein) \overset{def}{\equiv} & \left(\begin{array}{l} cat_N(act) \notin SP \land (mn'(act) = 0) \\ \land\ \triangle\,(\{act, ein\} \cup outtype(n, SF)\ ) \end{array} \right) & \\
& \lor \left(\begin{array}{l} cat_N(act) = SP \land mn(act) = 1 \\ \land \neg mayComplete(act) \land (mn'(act) = 0) \\ \land\ \forall nn \in R(act) \cap N, (mn'(nn) = 0) \\ \land\ \forall ee \in R(act) \cap E, (me'(ee) = 0) \\ \land\ \triangle\,(\{act, ein\} \cup R(act) \cup outtype(n, SF)) \end{array} \right) & \\
\forall n \in N, St(n) \overset{def}{\equiv} & cat_N(n) \notin MBE & \\
& \land\ \exists act \in N, cat_N(act) \notin A, (act = attachedTo(n)) \land (mn(act) = 1) & \\
& \land\ \exists ein \in intype(n, MF), (me(ein) \geq 1) & \\
& \quad \land\ receive(procOf(source(ein)), procOf(n), msg_t(ein)) & \\
& \quad \land\ (me'(ein) = me(ein) - 1) & \\
& \quad \land\ \forall eo \in outtype(n, SF), (me'(eo) = me(eo) + 1) & \\
& \quad \land \left(\begin{array}{l} \quad (isInterrupt(n) \land St_{interrupting}(n, act, ein)) \\ \lor\ (\neg isInterrupt(n) \land \triangle(\{ein\} \cup outtype(n, SF))) \end{array} \right) &
\end{array}
$$

Example. Figure 4.24 presents a part of a process with a none start event nse, an abstract task $task1$ with an outgoing sequence flow edge $e2$, an interrupting boundary event ($interrupt$) with an outgoing sequence flow edge $e4$, and two abstract tasks $task2$, and $task3$. The $interrupt$ boundary node starts by consuming a token from the activity it is attached to, receiving a message $m1$ from the network, and producing a token on its outgoing edge $e4$.

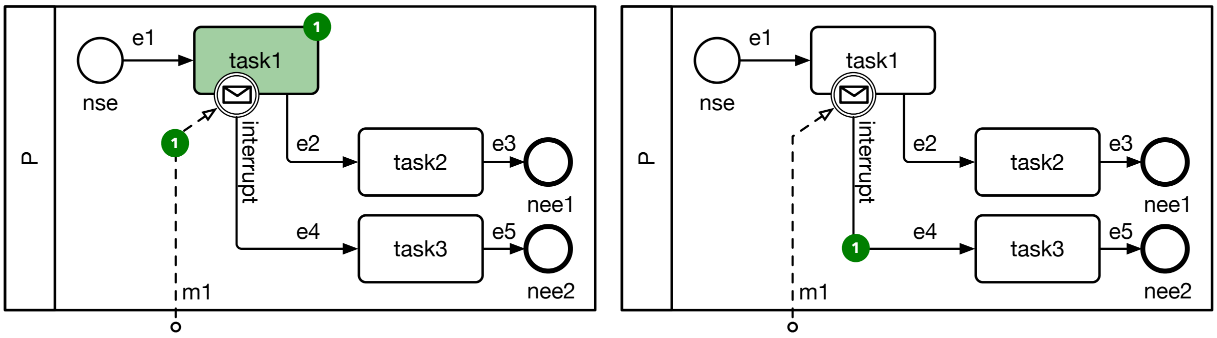


Figure 4.24: *Starting Behaviour of an Interrupting Message Boundary Event (task case). Before (left) and after (right) application of the St rule.*

Figure 4.25 shows an interrupting boundary event $interrupt$ attached to a sub-process SP. The $interrupt$ event interrupts the execution of SP when it receives a message $m1$. It cancels the execution of the sub-process by removing all its token (the token on it and the token on node $nse1$), and generates a token on its outgoing sequence flow edge, $e4$.

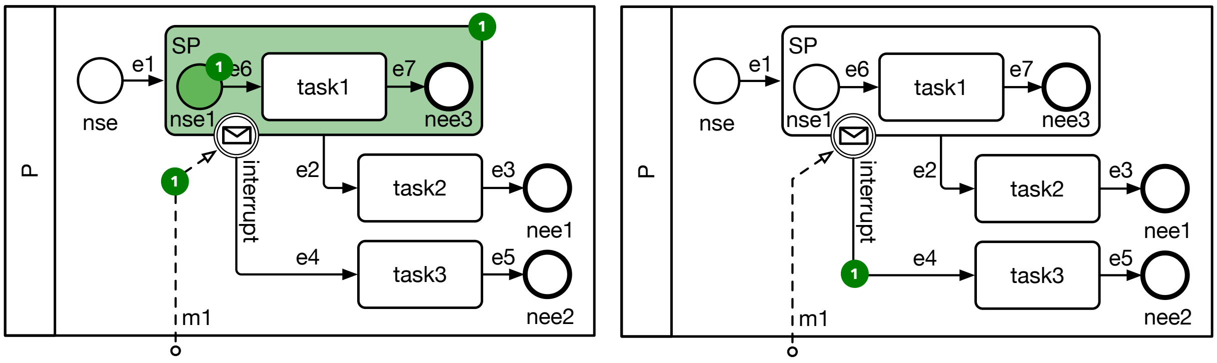


Figure 4.25: *Starting Behaviour of an Interrupting Message Boundary Event (subprocess case). Before (left) and after (right) application of the St rule.*

Timer Event The Timer Events defines three types the timer start event (TSE), the timer intermediate catch event ($TICE$), and the timer boundary event (TBE). As the semantics asynchronous, the TSE behaves precisely as a None Start Event (NSE), where the event will fire at some point. The $TICE$is indistinguishable from a gateway with one input and one output (it fires at some point). Formally:

$$\begin{aligned}\forall n \in N, Ct(n) \overset{def}{\equiv}\ & (cat_N(n) = TICE)\\ & \exists ei \in intype(n, ()SF), (me(ei) \geq 1) \wedge (me'(ei) = me(ei) - 1)\\ & \wedge\ \exists eo \in outtype(n, SF), (me'(eo) = me(eo) + 1)\\ & \quad \wedge \triangle\ (\{ei, eo\}) \wedge \Xi\end{aligned}$$

2370 The TBE is non-deterministically activated, without fairness (it may never fire). Its semantics is close to a MBE, without the constraint on the presence of a message. Formally:

$$\begin{aligned}St_{interrupting}(n, act) \overset{def}{\equiv}\ & (cat_N(act) \notin SP) \wedge (mn'(act) = 0) \wedge \triangle(\{act\} \cup outtype(n, SF)))\\ & \vee \left(\begin{array}{l}(cat_N(act) \in SP) \wedge \neg mayComplete(act) \wedge (mn'(act) = 0)\\ \wedge\ \forall nn \in R(act) \cap N, (mn'(nn) = 0)\\ \wedge\ \forall ee \in R(act) \cap E, (me'(ee) = 0)\\ \wedge \triangle\ (\{act\} \cup R(act) \cup outtype(n, SF))\end{array}\right)\end{aligned}$$

$$\begin{aligned}\forall n \in N^{TBE}, St(n) \overset{def}{\equiv}\ & \exists act \in N, (cat_N(act) \in A), (act = attachedTo(n)) \wedge (mn(act) = 1)\\ & \wedge\ \forall eo \in outtype(n, SF), (me'(eo) = me(eo) + 1)\\ & \wedge \left(\begin{array}{l}\quad (isInterrupt(n) \wedge St_{interrupting}(n, act))\\ \vee\ (\neg isInterrupt(n) \wedge \triangle(outtype(n, SF)))\end{array}\right)\\ & \wedge\ \Xi\end{aligned}$$

Gateways Gateways are atomic and define only the completing behaviour.

A parallel gateway (AND) is ready to complete if it has at least one token on all its incoming edges. It completes by removing one token on each of these edges and producing one on all its outgoing edges. **Formally.**

$$\begin{aligned}\forall n \in N, Ct(n) \overset{def}{\equiv}\ & (cat_N(n) = AND)\\ & \forall ei \in intype(n, SF), (me(ei) \geq 1) \wedge (me'(ei) = me(ei) - 1)\\ & \wedge\ \forall eo \in outtype(n, SF), (me'(eo) = me(eo) + 1)\\ & \wedge \triangle\ (intype(n, SF) \cup outtype(n, SF)) \wedge \Xi\end{aligned}$$

2375 **Example.** Figure 4.26 shows that the parallel gateway $AND1$ completes by consuming a token from its incoming edge ($e1$) and producing a token on all its outgoing edges ($e2$ and $e3$).

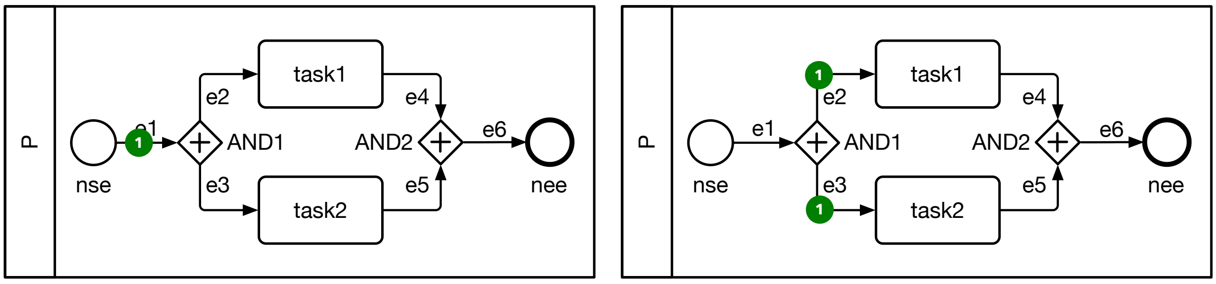


Figure 4.26: *Completing Behaviour of a Splitting Parallel Gateway. Before (left) and after (right) application of the Ct rule.*

In Figure 4.27, the parallel gateway $AND2$ completes only if all its incoming sequence flows edges ($e4$ and $e5$) are synchronised (*i.e.*, own at least a token).

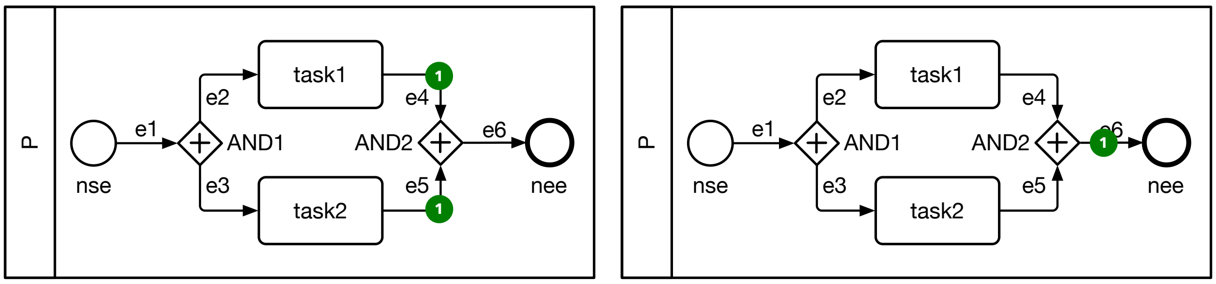


Figure 4.27: *Completing Behaviour of a Merging Parallel Gateway. Before (left) and after (right) application of the Ct rule.*

An exclusive gateway (XOR) is ready to complete if it has at least one token on one of its incoming
2380 edges. It completes by removing this token and producing one on one of its outgoing edges, depending on conditions. Since we abstract away from data, the concerned edge is non-deterministically chosen.
Formally.

$$\begin{aligned}\forall n \in N, Ct(n) \overset{def}{\equiv}\ & (cat_N(n) = XOR)\\ & \wedge\ \exists ei \in intype(n, SF), (me(ei) \geq 1)\\ & \wedge\ (me'(ei) = me(ei) - 1)\\ & \wedge\ \exists eo \in outtype(n, SF), (me'(eo) = me(eo) + 1)\\ & \wedge\ \triangle(\{ei, eo\}) \wedge \Xi\end{aligned}$$

Example. Figure 4.28 shows that the exclusive gateway ($XOR1$) completes by consuming a token from its incoming edge ($e1$) and producing a token on one of its outgoing edges ($e2$ in the example).

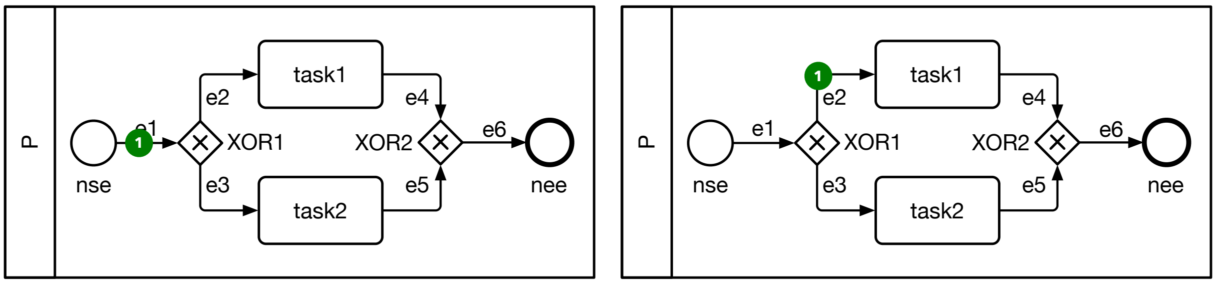
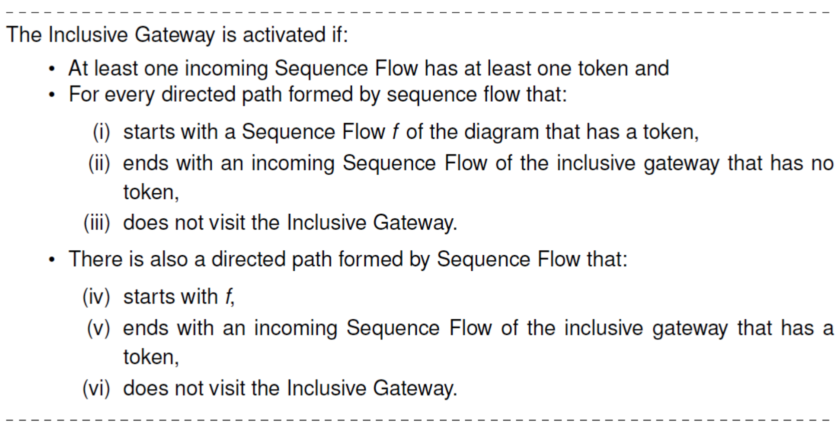


Figure 4.28: *Completing Behaviour of an Exclusive Gateway. Before (left) and after (right) application of the Ct rule.*

As described in the standard [3], an event-based gateway (EB) is always followed by communication
2385 elements, either receive tasks (RT) or intermediate catching message events ($CMIE$) in combination with intermediate catching timer event ($TICE$). The firing of an event-based gateway relies on the enabling of one of these elements. Hence, an event-based gateway completes by consuming a token from one of its incoming edges and producing a token on its outgoing edge on which the event is enabled.
Formally.

$$\begin{aligned}\forall n \in N, Ct(n) \overset{def}{\equiv}\ & (cat_N(n) = EB)\\ & \wedge\ \exists ei \in intype(n, SF), (me(ei) \geq 1) \wedge (me'(ei) = me(ei) - 1)\\ & \wedge \left(\begin{array}{l} \left(\begin{array}{l} \exists eo \in outtype(n, SF), cat_N(target(eo)) \in \{RT, CMIE\} \\ \wedge\ \exists em \in intype(target(eo), MF), (me(em) \geq 1) \end{array} \right) \\ \vee\ (\exists eo \in outtype(n, SF), cat_N(target(eo)) \in \{TICE\}) \end{array} \right)\\ & \wedge\ (me'(eo) = me(eo) + 1)\\ & \wedge\ \triangle(\{ei, eo\}) \wedge \Xi\end{aligned}$$

2390 **Example.** Figure 4.29 shows a process that contains an event-based gateway (EBG), and two receive tasks ($Rec.1$ and $Rec.2$). The left-hand side of the figure shows that the event-based gateway is enabled because it has a token on its incoming edge ($e1$) and an incoming message for at least one of the two receive tasks. This is true for both receive tasks here. Hence, EBG completes by consuming the token from $e1$ and generating a token on one of its two outgoing edges (arbitrarily chosen, here $e2$).



The Inclusive Gateway is activated if:

- At least one incoming Sequence Flow has at least one token and
- For every directed path formed by sequence flow that:
  - (i) starts with a Sequence Flow *f* of the diagram that has a token,
  - (ii) ends with an incoming Sequence Flow of the inclusive gateway that has no token,
  - (iii) does not visit the Inclusive Gateway.
- There is also a directed path formed by Sequence Flow that:
  - (iv) starts with *f*,
  - (v) ends with an incoming Sequence Flow of the inclusive gateway that has a token,
  - (vi) does not visit the Inclusive Gateway.

Figure 4.30: *Semantics of Inclusive Gateway According to the BPMN 2.0 Standard. (from source text [3])*

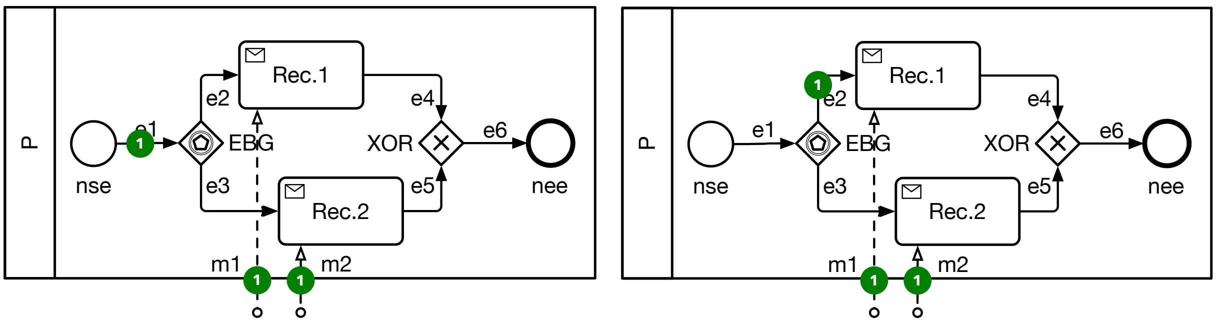


Figure 4.29: *Completing Behaviour of an Event-Based Gateway. Before (left) and after (right) application of the Ct rule.*

An inclusive gateway (OR) behaves differently from the other gateways. The activation of an OR gateway g is more complex [3, Chap. 13]. Figure 4.30 shows the standard semantics definition of the OR gateway. It is clear that this gateway has non-local semantics, and its activation may depend on the marking evolution considering the whole diagram. In more details, it can be activated only if:

- (1) it has at least one token on one of its incoming edges, and
- (2) for each marked node or edge x such that there is a path – that does not pass through g – from x to an unmarked incoming edge of g, there must also be a path – that does not pass through g – from x to a marked incoming edge of g.

The OR gateway completes by adding a token either to the outgoing edges whose conditions are true, otherwise to its default sequence flow edge. Since we abstract from data, we chose non-deterministically to add a token either to a combination (1 or more) of the outgoing non-default edges or to the default edge.

Auxiliary functions To formalise the semantics of an OR gateway, we define some auxiliary functions.

- $Pre_N : N \times E \rightarrow 2^N$, returns the predecessor nodes of an edge such that n^{pre} is in $Pre_N(n, e)$ if there is a path from n^{pre} to e that never visits n. Accordingly, $Pre_E : N \times E \rightarrow 2^E$, returns predecessor edges. These two sets can be structurally computed from the BPMN graph structure, hence can be taken as constants (for a given BPMN model).
- $InMinus$: $N \rightarrow E$, returns the unmarked incoming edges of a node:

$$InMinus(n) = \{e \in intype(n, SF) \mid me(e) = 0\}$$

- $InPlus : N \rightarrow E$, returns the marked incoming edges of a node:

$$InPlus(n) = \{e \in intype(n, SF) \mid me(e) \geq 1\}$$

- $ignore_E : N \rightarrow 2^E$, returns the set of predecessor edges of the marked incoming edges of a given
2415 node:

$$ignore_E(n) \overset{def}{\equiv} \bigcup_{e \in InPlus(n)} Pre_E(n, e)$$

- $ignore_N : N \rightarrow 2^N$, returns the set of predecessor nodes of the marked incoming edges of a given node:

$$ignore_N(n) = \bigcup_{e \in InPlus(n)} Pre_N(n, e)$$

Formally.

$$\begin{aligned}
\forall n \in N^{OR}, Ct(n) \overset{def}{\equiv}\ & (InPlus(n) \neq \emptyset) \\
& \wedge\ \forall e \in InPlus(n), (me'(e) = me(e) - 1) \\
& \wedge\ \forall ez \in InMinus(n), \forall ee \in (Pre_E(n, ez) \setminus ignore_E(n)), (me(ee) = 0) \\
& \wedge\ (\forall nn \in (Pre_N(n, ez) \setminus ignore_N(n)), (mn(nn) = 0)) \\
& \wedge \left(\begin{array}{l} \left(\begin{array}{l} \exists Outs \subset outtype(n, \{NSF, CSF\}), (Outs \neq \emptyset) \\ \quad \wedge\ \forall e \in Outs, (me'(e) = me(e) + 1) \\ \quad \wedge \bigtriangleup (InPlus(n) \cup Outs) \wedge \Xi \end{array} \right) \\ \vee \left(\begin{array}{l} \exists e \in out^{DSF}(n), (me'(e) = me(e) + 1) \\ \quad \wedge \bigtriangleup (InPlus(n) \cup \{e\}) \wedge \Xi \end{array} \right) \end{array} \right)
\end{aligned}$$

Example. Figure 4.31 illustrates the case when an OR getaway cannot be activated, despite a marked incoming edge ($e3$), there is a path from the marked edge $e2$ to an unmarked incoming edge of OR ($e6$ or $e7$) but no path from $e2$ to a marked incoming edge of OR. If the token on $e2$ had been on $e1$, the
2420 OR gateway could have been activated.

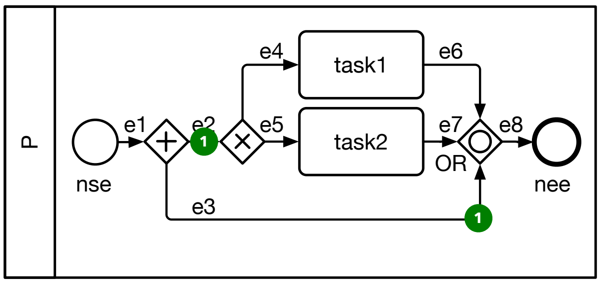


Figure 4.31: *Non Activable Inclusive Gateway. It has to wait for the token on $e2$ which is in $Pre_E(OR, e6)$ and $Pre_E(OR, e7)$.*

4.4.0.3 Communication

The properties of communication between two participants (process nodes) for a given type of message are abstracted with an initialisation function, *initnet*, and two transition predicates, *send* and *receive*. *initnet* is used to give the initial state of *mnet*. *send* and *receive* specify when a communication action is
2425 enabled and what effect it has on *mnet*. The value of *mnet* describes the state of the network in terms of what messages are sent but not yet received as the network evolves through time. In essence, we are modelling the pool of messages that have been sent but not yet received, possibly using a single universal queue, or by using channel-by-channel queues, or some other structures that carry information to allow and order send and receive events.

2430 Several communication models are formally described in Section 4.3.3. For instance, with the Fifo All asynchronous communication model, messages must be delivered in the order they were sent. In this model, $send(p_1, p_2, m)$ is always enabled, and adds m to $mnet$; $receive(p_1, p_2, m)$ is true only if m is the oldest message and thus the next one to be delivered, and the new state of $mnet$ is its previous value minus m.

Definition 4.4.3 (Communication Model)**.** The communication model is characterised by a function $initnet : T_{net}$ and two predicates *send*/*receive* defined above in 4.1.

4.4.0.4 Transition Relation and Executions

We can now expr.

Definition 4.4.4 (Transition Relation)**.** Let s and s' be two states. We say that s' is a successor of s, iff the predicate $Next(s, s')$ holds:

$$Next(s, s') \overset{def}{\equiv} \bigvee_{n \in N} (St(n) \vee Ct(n))$$

We recall that states, here s and s', correspond to tuples of the form (*me*, *mn*, *mnet*) and (*me'*, *mn'*, *mnet'*), whose elements are used in the definitions of St and Ct.

The execution of the whole process is defined through the notion of *trace*.

Definition 4.4.5 (Trace)**.** A trace is a finite or infinite sequence of states such that $\sigma[0]$ is the initial state, and $\forall i \in 0 \ldots Len(\sigma) - 1, Next(\sigma[i], \sigma[i+1])$ (if the trace is finite) or $\forall i \in \mathbb{N}, Next(\sigma[i], \sigma[i+1])$ (if the trace is infinite), where $\sigma[i]$ denotes the i^{th} state of the trace. The set of all the traces of the collaboration is noted $Traces$.

Definition 4.4.6 (Execution)**.** An execution is a maximal trace, *i.e.*, a trace that goes as far as possible. Formally, an execution σ is either an infinite trace, or a finite trace such that $\neg\exists s', Next(\sigma[Len(\sigma)], s')$. The set of all the executions of the collaboration is noted $\mathcal{E}xec$.

4.4.0.5 Fair Executions

A BPMN model can include loops. In that case, an execution (defined as a maximal trace) can get stuck in a loop, where only this loop progresses, and the rest of the model doesn't progress at all. Moreover, when modelling actual business activities, loops are expected to finish at some point. To prevent these infinite loops, fairness is introduced to restrain the set of executions. We use two kinds of fairness: *weak fairness* and *strong fairness*. Informally, weak fairness ensures that a transition cannot be permanently enabled and never fired. Strong fairness ensures that a transition cannot be infinitely often enabled and never fired.

Thus, fairness is a conjunction of two parts. The first part is the weak fairness on each start (St) and complete (Ct) transitions of every node: $\forall n \in Node : weakfair(step(n))$. This property ensures that any permanently enabled transition eventually occurs. This means that no process may progress forever while others are never allowed to do so if they can. This also means that if a process contains several loops that are simultaneously live, all loops will progress (not necessarily at the same speed, but no loop can be permanently halted while another run forever).

The second part is the strong fairness on each output edge of XOR, OR, and EB gateways. Strong fairness ensures that no choice is infinitely often ignored: if a XOR, OR, or EB gateway is included in a loop, the fairness forbids the infinite executions that neveess the complete transition relation (successor relation between states) with the previously defined predicates. If some output edges. Either the loop finishes somehow, or all the choices are infinitely often taken. Consider Figure 4.32, left; as strong fairness is imposed on the two output edges of gateway *choice*, the execution cannot always ignore the edge ($e5$) leading to the ending node, and this model is sound. Consider Figure 4.32, right; as strong fairness is imposed on the output edges of gateway *choice*, both $task1$ and $task2$ are infinitely often chosen.

4.5 Verification Properties

Verifying a model involves checking the correctness of its properties. In the context of process modelling, properties are classified into two main classes: *structural* and *behavioural*. The *structural properties* relate to the type of elements and how they are connected. Such properties could be checked using a standard process modelling tool which can enforce that the model is correctly designed. The *behavioural properties* relate to the sequences of execution as defined by the process model. We further classify the behavioural properties into *general* and *specific* ones. The specific properties are unique to business process models, while the general properties are used in other types of models.

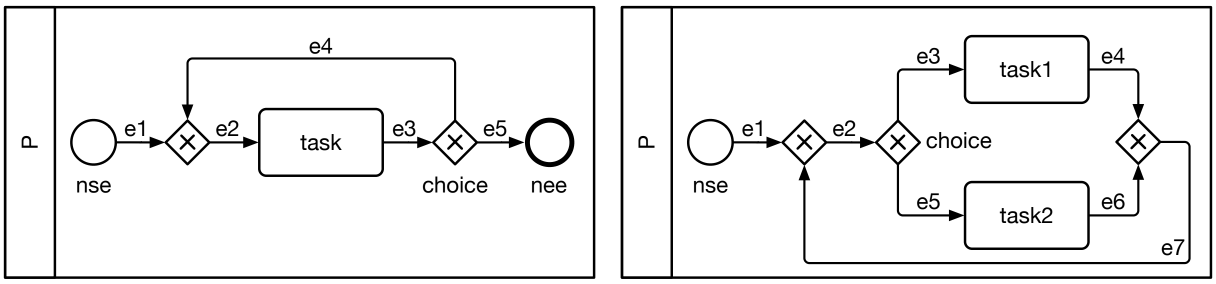


Figure 4.32: *Use of Strong Fairness to Avoid Infinite Loops (left) and Starvation (right).*

General properties *deadlocks* and *livelocks* are common examples of general properties. Figure 4.33 shows a simple BPMN model with an XOR and an AND gateways. The XOR gateway produces a token either on its outgoing edge $e2$ or $e3$ but not on both. As a consequence, the AND gateway will never be enabled (its incoming edges $e4, e5$ will never be synchronised). Hence, this model suffers a deadlock situation. While in a deadlock, the involved activities can never be executed, and the process can never be completed. In a livelock situation, a set of activities are executed indefinitely.

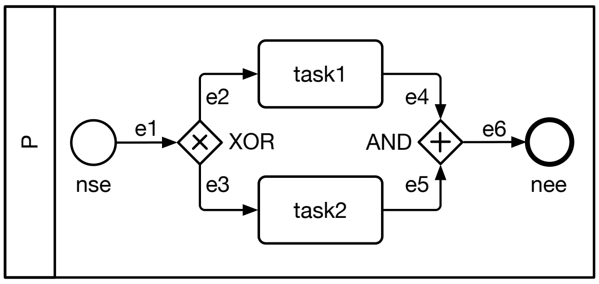


Figure 4.33: *BPMN Diagram with Deadlock.*

Specific properties *Soundness* is the leading property that can be checked after a process model is executed.

Wil van der Aalst developed soundness property for business processes in the context of workflow nets [191]. A workflow net has a unique terminal place. The authors defined the soundness of the WF-net by the satisfaction of the three following requirements: "*(1) Option to complete: from any reachable state, it is possible to reach a state with marks on the terminal place, (2) proper completion: if the terminal place is marked, all other places are empty, (3) no dead transitions: it should be possible to execute an arbitrary activity by following an appropriate route through the WF-net*". In the case of Petri nets in workflow verification, [177], and [81] prove that a workflow net is sound if and only if the corresponding short-circuited Petri net is live and bounded. In addition, they define the safeness of a WF-net by ensuring that each place cannot hold multiple tokens at the same time.

Dumas et.*al* [1, p.186, ch.5] define informally the soundness of the BPMN process model by the satisfaction of the following three properties: "*(1) Option to complete: any running process instance must eventually complete, (2) Proper completion: at the moment of completion, each token of the process instance must be in a different end event, (3) No dead activities: any activity can be executed in at least one process instance*".

In [23], authors address BPMN collaboration models. They introduce a formal definition of safeness and soundness properties by focussing on the specificities of BPMN. "*A process is safe if during its execution no more than one token occurs along the same sequence edge*". The authors extend the safeness property for processes collaboration which require that "*each of all the processes that involved in the overall collaboration execution is safe*". On the other hand, they define the soundness of the process as follows: "*A BPMN process is sound if it can complete its execution without leaving active elements and all the model elements can be activated in at least one of the execution traces*". In addition, they extend the latter to define the soundness of the whole collaboration model.

Based on those definitions [1, 23], we provide a set of properties that can be formally specified as follows. We recall here that a state in our formalisation of BPMN is $s = (mn, me, mnet)$. For an execution $\sigma = s_1 s_2 ...$ and a node x, we note $\sigma[i].mn(x)$ the value of the marking of x at step i in σ, and

$domain(\sigma) = \{1, ..., |\sigma|\}$.

Definition 4.5.1 (Option to complete)**.** Any running process must eventually be complete. A process is complete in a state if markings occur only on end events.

$$\begin{aligned} Completed(p,s) \overset{def}{\equiv} \;& \forall n \in R(p) \cap N, (s.mn(n) = 0) \vee (s.mn(n) = 1 \wedge n \in N, cat_N(n) = EE) \\ & \wedge\, (\forall e \in R(p) \cap E, (s.me(e) = 0)) \end{aligned}$$

$$\begin{aligned} OptionToComplete \overset{def}{\equiv} \;& \forall p \in N, cat_N(p) = P, \forall \sigma \in \mathcal{E}xec, \forall i \in domain(\sigma), \sigma[i].mn(p) > 0 \Rightarrow \\ & \exists j \in domain(\sigma), j \geq i \wedge Completed(p, \sigma[j]) \end{aligned}$$

Definition 4.5.2 (Proper Completion)**.** At the moment of completion, each token of the process instance must be in a different end event.

$$\begin{aligned} ProperCompletion \overset{def}{\equiv} \;& \forall p \in N, cat_N(p) = P, \forall \sigma \in \mathcal{E}xec, \forall i \in domain(\sigma), \\ & Completed(p, \sigma[i]) \Rightarrow \forall n \in R(p), cat_N(n) = EE, \sigma[i].mn(n) = 1 \end{aligned}$$

Definition 4.5.3 (No dead activities)**.** An activity is dead if no execution activates it.

$$NoDeadActivities \overset{def}{\equiv} \forall a \in N^A, \exists \sigma \in \mathcal{E}xec, \exists i \in domain(\sigma), \sigma[i].mn(a) \neq 0$$

Definition 4.5.4 (Undelivered messages)**.** No messages are left in transit.

$$\begin{aligned} NoUndeliveredMessages \overset{def}{\equiv} \;& \forall \sigma \in \mathcal{E}xec, \exists i \in domain(\sigma), \forall j \in domain(sigma), \\ & j \geq i \implies \forall e \in MF, \sigma[j].me(e) = 0 \end{aligned}$$

Definition 4.5.5 (Safe process)**.** A process is safe if and only if all its sequence flow edges never hold more than one token during their execution.

$$\text{For } p \in N, cat_N(p) = P, SafePr(p) \overset{def}{\equiv} \forall \sigma \in \mathcal{E}xec, \forall i \in domain(\sigma), \forall e \in R(p) \cap E, (\sigma[i].me(e) \leq 1)$$

Definition 4.5.6 (Safe Collaboration)**.** A collaboration is safe all its processes are safe.

$$Safe \overset{def}{\equiv} \forall p \in N, cat_N(p) = P, SafePr(p)$$

Definition 4.5.7 (Process soundness)**.** A process is a sound in a state if only its end events hold at most one token, and all the other nodes (ignoring start events) and all the edges are unmarked. Formally, process $p \in N, cat_N(p) = P$ is sound in a state s if and only if the following predicate is true :

$$\begin{aligned} soundPr(p,s) \overset{def}{\equiv} \;& \forall n \in R(p) \cap N, (s.mn(n) = 0) \vee (s.mn(n) = 1 \wedge cat_N(n) \in \{EE, SE\} \\ & \wedge\, \forall e \in R(p) \cap E, s.me(e) = 0 \end{aligned}$$

Definition 4.5.8 (Message-relaxed sound collaboration)**.** A collaboration is message-relaxed sound if eventually all the processes are sound and it is stable:

$$\begin{aligned} msgSoundCol \overset{def}{\equiv} \;& \forall \sigma \in \mathcal{E}xec, \exists i \in domain(\sigma), \forall j \in domain(\sigma), \\ & j \geq i \implies \forall p \in N, cat_N(p) = P, soundPr(p, \sigma[j]) \end{aligned}$$

Definition 4.5.9 (Collaboration soundness)**.** A collaboration is sound if and only if, for all executions, eventually, all the processes involved in the collaboration are sound, and all the message flow edges are unmarked.

$$\begin{aligned} soundCol \overset{def}{\equiv} \;& \forall \sigma \in \mathcal{E}xec, \exists i \in domain(\sigma), \forall j \in domain(\sigma), j \geq i \\ & \implies \forall p \in N, cat_N(p) = P, soundPr(p, \sigma[j]) \\ & \quad \wedge \quad \forall e \in E, cat_E(e) = MF, (\sigma[j].me(e) = 0) \end{aligned}$$

Regarding other definitions of soundness [1, 23, 191], we consider a form of soundness under fairness assumptions, that could be called *fair soundness*.

4.6 Summary

This chapter proposes a direct formalisation in first-order logic for a subset of BPMN that includes subprocess and communication elements. We integrate a communication channel with seven communication models to parametrise the verification regarding these models. To better illustrate their impact on the properties of a BPMN model, Figure 4.34 shows a simple Travel agency collaboration model, where the communication is given by two participants, a customer and a travel agency. This collaboration represents an unsafe travel agency process: (1) it can have an unbounded number of tokens on the right of the parallel gateway; (2) observe that the partners disagree on the order of confirmation w.r.t. ticket reception. Depending on the communication model, e.g., the FIFO model choice in this model may cause a deadlock, whereas the Bag model choice removes this deadlock.

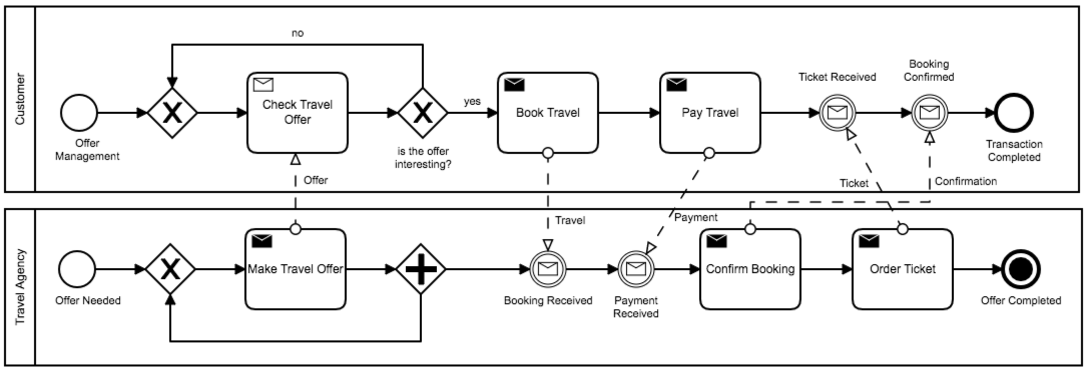


Figure 4.34: *Travel Agency Case Study. (Slightly adapted from an example in [23])*

Indeed, we have adopted the message relaxed property that highlights the non consumed message issue that the interactions may bring. For example, Figure 4.35 shows a collaboration example between two processes, a client and a worker. This model identifies a possible undesired behaviour where the client and the worker processes may complete correctly without a deadlock. At the same time, a cancellation message is still present on the communication channel. By conceding the message relaxed property, these processes may be seen as sound processes. Identifying such cases may become increasingly difficult and interesting when considering larger and more complex models.

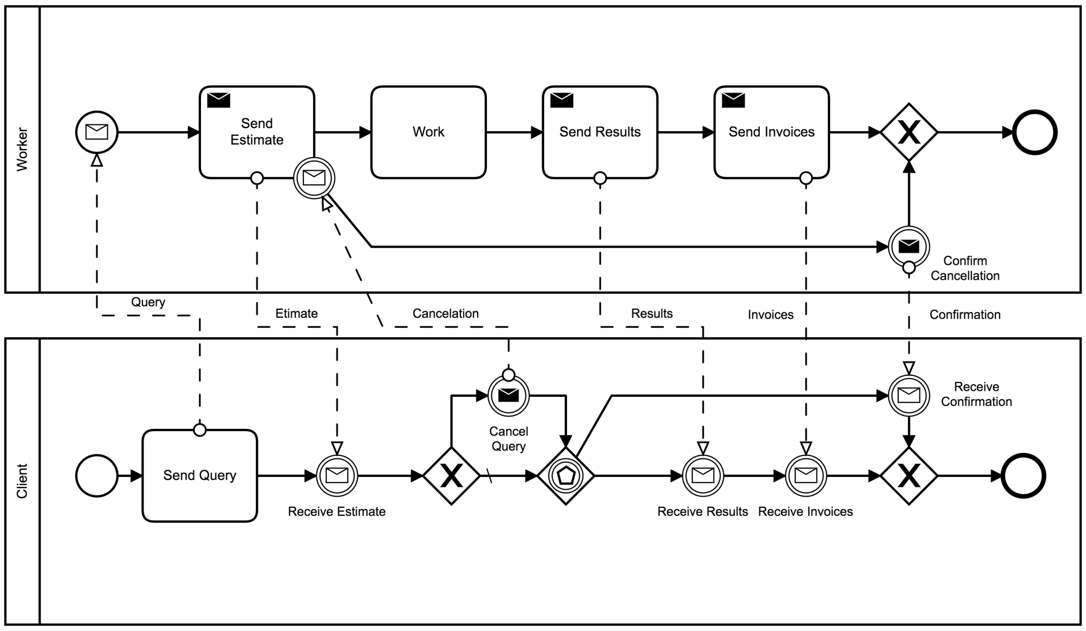


Figure 4.35: *Client/Worker Case Study.*

Also, the proposed formalisation covers the hierarchical structure by supporting subprocess elements.

These elements haven't only syntactic influence, but their presence on the process models also impact their correctness verification. To illustrate the latter, we consider Figure 4.36 and Figure 4.37. They present an extension of the running example scenario presented in Figure 2.9. For the sake of presentation, only the Journal Chair participant is reported. Figure 4.36 shows a process with prepare notification activities embedded into a subprocess, where Figure 4.37 flattens these activities into the main process. Since the property of soundness is the most commonly requested quality criteria for business processes. These two processes lead to different results with reference to this property. Figure 4.37 results on an unsound process, while the process of Figure 4.36 results on an unsound subprocess with a sound process since only one token is produced on the outgoing of the subprocess after its execution.

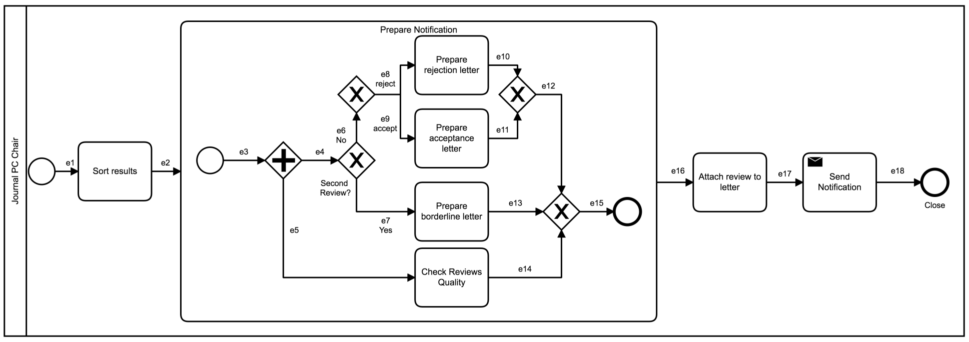


Figure 4.36: *Journal chair Process (Sound).*

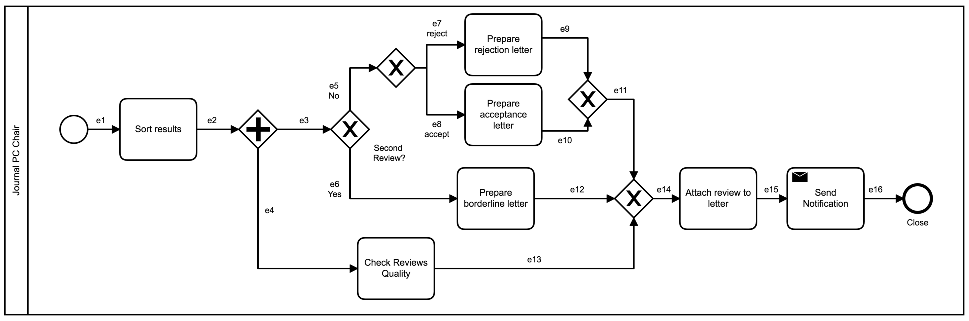


Figure 4.37: *Journal chair Process (UNSound).*

A limitation of the proposed formal semantics and an avenue for future work arise because the formal semantics does not consider the timer elements with an explicit time notion. As an example for this limitation, TBE can never be fired in BPMN semantics (*e.g.*, a date in the past): the non-deterministic semantics allows it to fire. This gives us an over-approximation: the non-deterministic semantics contain the same executions as BPMN semantics, plus additional ones. Thus, if the verification states that a property is verified with the non-deterministic semantics, it is necessarily verified with BPMN semantics. The reverse is not true. For instance, in the example of Figure 4.38, BPMN semantics states that the task, $task2$, should never be activated as its time constraint is always in the past. The non-deterministic semantics defines two executions: (i) the timeout will not fire, (ii) the timeout will fire, and the $task2$ will be activated.

For that, some challenging issues, like *"How can temporal constraints be formalised?"* and *"What happens if a temporal condition can no longer be satisfied ?"*, are picked up in the following chapter.

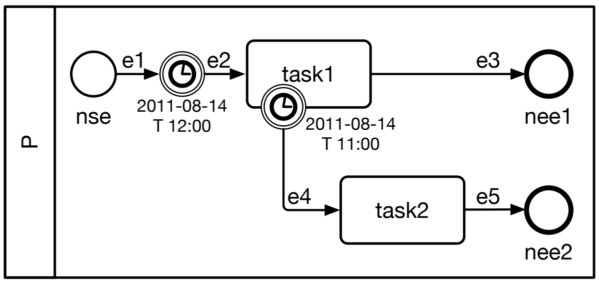


Figure 4.38: *Timer Boundary Event with an Impossible Timeout. As the timeout on $task1$ is later than the date imposed by the timer intermediate catch event, this timeout should never fire and $task2$ should never be activated.*

Chapter 5

BPMN and Time

"You may delay, but time will not."
Benjamin Franklin

Chapter content

5.1 Introduction

Both the specification and the operational support of temporal constraints constitute fundamental challenges for any BPMS. To support the design and implementation of time-aware processes, a variety of temporal concepts (*e.g.*, deadlines, minimum and maximum time lags, durations, and schedules) need to be supported by the BPMS. Therefore, respective concepts have to be supported by the used process modelling language notation, *i.e.*, BPMN. For that, BPMN defines a set of time-related events: *Timer Start Events (TSE), Timer Intermediate Catch Events (TICE), and Timer Boundary Events (TBE)*, Interrupting or not (see 2.4). All these three kinds of events depend on a time-related condition defined in their *TimerEventDefinition* [3]. This, in turn, relies on the ISO-8601 standard [16] definitions. However, the resulting expressiveness of these BPMN timer related constructs hampers the definition of formal semantics, including them, and the provision of formal analysis means for timed process models.

To tackle the discussed challenges, this chapter presents, in complement to Chapter 4 where time was abstracted in a non-deterministic way, an explicit semantics for time notion in BPMN. In addition, we formally assess the suitability of BPMN to support the process *Time Patterns* were introduced in [15].

BPMN Standard			TSE	TICE	TBE	TBE
TimerEventDefinition	ISO-8601				Interrupt	non-Interrupt
timeDate	date and time	yyyy-mm-ddThh:mm:ssZ	○	○	○	○
timeCycle	unbounded	R/ yyyy-mm-ddThh:mm:ssZ / yyyy-mm-ddThh:mm:ssZ	○	○	−	○
		R/ yyyy-mm-ddThh:mm:ssZ / PnYnMnDTnHnMnS	○	○	−	○
		R/ PnYnMnDTnHnMnS/yyyy-mm-ddThh:mm:ssZ	○	○	−	○
		R/PnYnMnDTnHnMnS	○	○	−	○
	bounded	Rn/ yyyy-mm-ddThh:mm:ssZ/yyyy-mm-ddThh:mm:ssZ	○	○	−	○
		Rn/ yyyy-mm-ddThh:mm:ssZ/PnYnMnDTnHnMnS	○	○	−	○
		Rn/ PnYnMnDTnHnMnS/yyyy-mm-ddThh:mm:ssZ	○	○	−	○
		Rn/ PnYnMnDTnHnMnS	○	○	−	○
timeDuration	duration	PnYnMnDTnHnMnS	−	○	○	○

Table 5.1: *Time-related Features in BPMN and their Relation to the ISO-8601 Standard. (○) supported category in BPMN, (−) not supported .*

This chapter is organised as follows. Section 5.2 introduces the support for the different BPMN timer elements using a typed graph. Section 5.3 gives a formal semantics to BPMN timer events based on an explicit time notion. To check whether our semantics is consistent, we study its support for the set of process time patterns provided in the literature in Section 5.4. Then, a summary is given in Section 5.5.

5.2 A Typed Graph Representation of BPMN Time-Related Constructs

To integrate the notion of explicit time in our formalisation, we extend the BPMN graph structure given in 4.2.1 to deal with some time constraints.

BPMN Time Elements. The BPMN standard associates three categories of time to the timer nodes:

- *timeDate* specifies a fixed date and time;
- *timeCycle* specifies repeating intervals;
- *timeDuration* specifies the amount of time a timer should run before firing.

Table 5.1 gives a synthetic view of the time-related events in BPMN with reference to these three categories. To support the latter, we define the time categories formally, $Ctime = \{T_{date}, T_{duration}, T_{cycle}\}$, and the following time structures to characterize the time constraints, $timeVal = Date \cup Duration \cup Cycle$:

- $Date \subseteq \mathbb{N}$ represents a date (and time) expressed in seconds with respect to a reference date (1970-01-01 T 00:00:00Z). *Date* refers to the timeDate of the BPMN standard. A date like 2020-12-03 T 13:52:33Z in ISO-8601 format is converted to $1,607,003,553$ seconds.
- $Duration \subseteq \mathbb{N}$ represents a time duration in seconds. This corresponds to the timeDuration of the BPMN standard. For example, a P3DT15M duration in ISO-8601 format (three days and fifteen minutes) is converted to $259,215$ seconds. Note that we do not support years and months in durations due to the ambiguity of their correspondence in seconds.
- $Cycle = (\mathbb{N} \cup \{\iota\}) \times [Duration \cup (Date \times Duration) \cup (Duration \times Date)]$, represents a composite timing type. It defines time recurrence along with time duration, fixed start date and time duration, or a time duration and a fixed end date. The number of repetitions is either bounded or not (ι). For example, a cycle with timeDate and Duration in ISO-8601 format R2/2020-02-01 T 00:00:00Z/P15D (Two recurrences between fifteen days starting from 2020-02-01 T 00:00:00Z date) is converted to $R2/1580511600/1296000$.

Taking into account these categories, we redefine the set of timer element types given in 4.2.1 as follows.

- The set (TSE) of timer start event types, groups the start event with time date category (TSE_d[1])

[1]In TSE_d, d stands for *timeDate* as in the ISO-8601 standard for time and date.

and the start event with time cycle category (TSE_c[2]). Formally:

$$TSE = \{TSE_d, TSE_c\}$$

- The set ($TICE$) of timer catch event types, groups the timer catch event with time date category ($TICE_d$) and the timer catch event with time duration category ($TICE_p$[3]). Formally:

$$TICE = \{TICE_d, TICE_p\}$$

- The set (TBE) of timer boundary event types, groups the interrupting ($TBE^{\oslash}$) and the non-interrupting ($TBE^{\oplus}$) boundary event sets $TBE = TBE^{\oslash} \cup TBE^{\oplus}$, with :
  - The set ($TBE^{\oslash}$) of interrupting timer boundary event types, groups the timer boundary event with time date category ($TBE_d^{\oslash}$) and the timer boundary event with time duration category ($TBE_p^{\oslash}$). Formally:

$$TBE^{\oslash} = \{TBE_d^{\oslash}, TBE_p^{\oslash}\}$$

  - The set ($TBE^{\oplus}$) of non-interrupting timer boundary event type, groups non-interrupting timer boundary event with time date category ($TBE_d^{\oplus}$), the non-interrupting timer boundary event with time duration category ($TBE_p^{\oplus}$), and the non-interrupting timer boundary event with time cycle category ($TBE_c^{\oplus}$). However, the time cycle category ($TBE_c^{\oplus}$) may be defined based on a bound or unbound number of recurrences with some specifications: start date and end date, start date and duration ($TBE_{c(start)}^{\oplus}$), duration and end date ($TBE_{c(start)}^{\oplus}$), only a duration ($TBE_{c(p)}^{\oplus}$). Formally: $TBE_c^{\oplus} = \{TBE_{c(start)}^{\oplus}, TBE_{c(start)}^{\oplus}, TBE_{c(p)}^{\oplus}\}$. Therefore, the set of $TBE^{\oplus}$ is defined formally as follows:

$$TBE^{\oplus} = \{TBE_d^{\oplus}, TBE_p^{\oplus}\} \cup TBE_c^{\oplus}$$

BPMN Standard			TSE	TICE	TBE Interrupt	TBE non-Interrupt
TimerEventDefinition	ISO-8601					
timeDate	date and time	yyyy-mm-ddThh:mm:ssZ	●	●	●	●
timeCycle	unbounded	R/ yyyy-mm-ddThh:mm:ssZ / yyyy-mm-ddThh:mm:ssZ	○	○	–	○
		R/ yyyy-mm-ddThh:mm:ssZ / PnYnMnDTnHnMnS	○	○	–	●
		R/ PnYnMnDTnHnMnS/yyyy-mm-ddThh:mm:ssZ	○	○	–	●
		R/PnYnMnDTnHnMnS	○	○	–	●
	bounded	Rn/ yyyy-mm-ddThh:mm:ssZ/yyyy-mm-ddThh:mm:ssZ	○	○	–	○
		Rn/ yyyy-mm-ddThh:mm:ssZ/PnYnMnDTnHnMnS	○	○	–	●
		Rn/ PnYnMnDTnHnMnS/yyyy-mm-ddThh:mm:ssZ	○	○	–	●
		Rn/ PnYnMnDTnHnMnS	○	○	–	●
timeDuration	duration	PnYnMnDTnHnMnS	–	●	●	●

Table 5.2: *Time-Related Features in BPMN and their Relation to the ISO-8601 Standard. Supported Category: BPMN and Us (●), BPMN Only (○), and Not Supported (–).*

Table 5.2 presents the timer nodes types and their timerDefinition categories supported in our work. Therefore, the formal definition of the nodes type sets presented in Section 4.2.1 is redefined as follows:

- The set of starting event types, $SE = \{NSE, MSE\} \cup TSE$
- The set of intermediate event types, $IE = \{CMIE, TMIE\} \cup TICE$
- The set of boundary event types, $BE = \{MBE\} \cup TBE$

In this chapter, we consider the two sets of basic elements types, T_{Nodes} and T_{Edges}, taking into account the updated sets SE, IE, and BE.

Notation. We use $Timer$ as a notation for the set of timer types, formally is defined follows:

$$Timer = TSE \cup TICE \cup TBE$$

[2]In TSE_c, c stands for *timeCycle* as in the ISO-8601 standard for time repeating intervals.

[3]In $TICE_p$, p stands for *timeDuration* as in the ISO-8601 standard for durations represented by P.

Definition 5.2.1 (BPMN (Timed) Graph)**.** (extended from Definition 4.2.1) A BPMN (timed) graph corresponds to the BPMN graph from Chapter 4 extended with a timing function: $\widehat{G}$= (N, E, $\mathbb{M}$, cat_N, cat_E, src, tgt, R, msg_t, $attachedTo$, $isInterrupt$, $ftime$) such that:

- $ftime : \{n \in N \mid cat_N(n) \in Timer\} \rightarrow Ctime \times timeVal$, associates a time category and a value to the timer nodes.

Notation. To denote the projection of the function $ftime$ on a component of its co-domain, we use the notation $\lfloor_{Ctime}$(resp. $\lfloor_{timeVal}$): for example, if $ftime(n) = (T, V)$, where $T \in Ctime$ and $V \in timeVal$, then $ftime(n) \lfloor_{Ctime}= T$, and $ftime(n) \lfloor_{timeVal}= V$. Besides, when $ftime(n) \lfloor_{Ctime}= T_{cycle}$, then $ftime(n) \lfloor_{timeVal}= (r, d, p)$, with $(r, d, p) \in Cycle$. The projections $\lfloor_{timeVal_R}$, $\lfloor_{timeVal_P}$, and $\lfloor_{timeVal_D}$ give each element, with $ftime(n) \lfloor_{timeVal_R}= r$, $ftime(n) \lfloor_{timeVal_D}= d$ and $ftime(n) \lfloor_{timeVal_P}= p$.

We stress that the BPMN models understudy must respect the well-formedness rules mentioned in Section 4.2.3.

5.3 A FOL Semantics for BPMN Time-Related Constructs

This section extends the formal semantics that we proposed in Section 4.4 to handle time constructs with associated ISO-8601 time information. We rely on a (typed) graph representation of the workflow and collaboration models where types correspond to kinds of BPMN elements as given above in Section 5.2.

To represent the global configuration of a BPMN model (workflow or collaboration) at any moment of its execution, we rely on the state notion. We extend the state definition with a global clock, a set of local clocks, and a recurrence function. Hence, the definitions for the state (Definition 4.4.1) and initial state (Definition 6.4) change as follows.

Definition 5.3.1 (State)**.** A state of a BPMN graph $\widehat{G}$= (N, E, $\mathbb{M}$, cat_N, cat_E, src, tgt, R, msg_t, $attachedTo$, $isInterrupt$, $ftime$) is denoted by a tuple $s = (m_n, m_e, mnet, l_c, g_c, rec)$ such that:

- $m_n : N \rightarrow \mathbb{N}$ and $m_e : E \rightarrow \mathbb{N}$, are marking functions, that associate a number of tokens to nodes and edges (respectively).
- $mnet : T_{net}$, is the state of the communication network.
- $l_c : \{n \in N \mid cat_N(n) \in Timer\} \rightarrow \mathbb{N}$, is a local clock whose value represent the time spent on a timer node.
- $g_c \in \mathbb{N}$, is a global clock representing the current time of the whole model.
- $rec : \{n \in N \mid cat_N(n) \in TBE_c^{\oplus}\} \rightarrow \mathbb{N} \cup \{\iota\}$ represents, for each activated non-interrupting timer boundary event node with a finite cycle, the number of occurrences that remains to be executed.

The set of all states of a BPMN graph is denoted by $States$.

Definition 5.3.2 (Initial state)**.** The initial state $s_o = (m_{n_0}, m_{e_0}, mnet_0, l_{c_0}, g_{c_0}, rec)$ of a BPMN graph is extended as follows:

- $m_{n_0}(n), m_{e0}(e), and\ mnet_0$ are initialised as before (Definition 6.4)
- The global clock is initialised to a specific date and time (w.r.t. a modelling referential[4]): $g_{c_0} \in \mathbb{N}$;
- Local clocks are initialised to zero: $\forall n \in N, cat_N(n) \in Timer$, l_{c_0}(n)=0;
- Redundancy variables are initialised with the recurrence number (if it exists) else 0:

$$\forall n \in N, cat_N(N) \in TBE_c^{\oplus}, rec_0(n) = ftime(n) \lfloor_{timeVal_R}$$

5.3.1 Semantics

Based on these changes, we define here the execution semantics of the timer event nodes based on those mentioned above St and Ct predicates. In the semantics, let $s = (mn, me, mnet, l_c, g_c, rec)$ and $s' = (mn', me', mnet', l'_c, g'_c, rec')$ denote two states. Additionally, we consider all the predicate introduced

[4]referential: an absolute date and time

in 4.4.0.2 (such as △(n), Ξ, *mayComplete*(n)) and we introduce two others as follows:

- *run* is a predicate that increases the local clock of each active timer events node and the global clock at once.

$$run() \overset{def}{\equiv} \forall n \in S, (l'_c(n) = l_c(n) + 1) \wedge \ (g'_c = g_c + 1) \tag{5.1}$$

- $\triangle_t$ is a predicate that denotes that clocks do not change except for the local ones for the nodes in X:

$$\triangle_t(X) \overset{def}{\equiv} g'_c = g_c \wedge \forall n \in N^{Timer} \setminus X, l'_c(n) = l_c(n) \tag{5.2}$$

The formal semantics of the BPMN time-related constructs is given in the following.

Timer Start Event (TSE). As shown in Table 5.2, we support only the timer start event with date configuration (TSE_d). The behaviour of this event is defined only by a completing predicate. It is only enabled to complete if it has a token and the (global) clock has reached the given deadline date. It completes by initiating the process to which it belongs and generating a token on its outgoing edges.

$$\begin{aligned} Ct(n) \overset{def}{\equiv}\ & (cat_N(n) = TSE_d) \wedge (ftime(n) \downharpoonright_{timeVal_D} = g_c) \\ & \wedge (mn(n) = 1) \wedge (mn'(n) = mn(n) - 1) \\ & \wedge \forall eo \in outtype(n, SF), (me'(eo) = me(eo) + 1) \\ & \wedge \exists p \in N, cat_N(n) = P, n \in R(p), (mn(p) = 0) \wedge (mn'(p) = mn(p) + 1) \\ & \wedge \ \triangle(\{n, p\} \cup outtype(n, SF)) \wedge \triangle_t(\emptyset) \wedge \Xi \end{aligned} \tag{5.3}$$

Example. Figure 5.1 presents a process p with a timer start event (*start*) defined a fixed timeDate ($2021-04-16\ T\ 00:00:00\ Z$), an abstract task (*task*), and a none end event (*end*). The left part of the figure shows that the start node is enabled to complete: it owns a token, the process depends on is inactive (*i.e.*, not owns a token), and the global clock time of the model has a timeDate reference to ($2021-04-16\ T\ 00:00:00\ Z$). The *start* node completes by consuming this token and producing one on P and its outgoing sequence flow ($e1$).

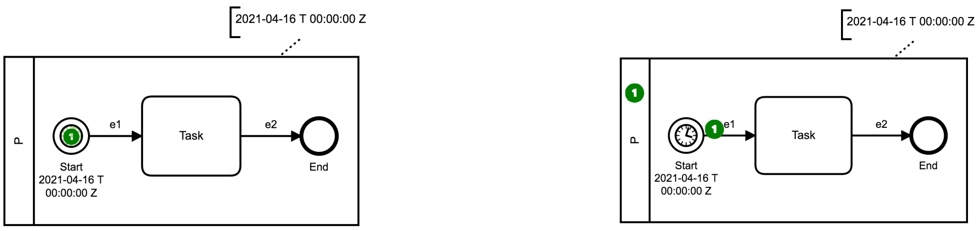


Figure 5.1: *Completing Behaviour of a Timer Start Event (Date). Before (left) and after (right) application of the Ct rule.*

Timer Intermediate Catch Event ($TICE$). Acts as a delay mechanism configured either by a duration ($TICE_p$), a fixed date ($TICE_d$), or a cycle (not supported in this work, see Table 5.2). Such an event waits for the specified date or duration before letting the control flow on which it is located continue.

More precisely, the behaviour of the $TICE_d$ event is defined by a completing predicate. It is enabled to complete if it has a token on one of its incoming edges. When the global clock reaches its fixed date, it completes by consuming a token from one of its incoming edges and generating one on all its outgoing edges. Formally:

$$\begin{aligned} Ct(n) \overset{def}{\equiv}\ & (cat_N(n) = TICE_d) \wedge (ftime(n) \downharpoonright_{timeVal_D} = g_c) \\ & \wedge \exists e \in intype(n, SF), (me(e) = 1) \wedge (me'(e) = me(e) - 1) \\ & \wedge (\forall e' \in outtype(n, SF), (me'(e') = me(e') + 1)) \\ & \wedge \ \triangle(\{e\} \cup outtype(n, SF)) \wedge \triangle_t(\emptyset) \wedge \Xi \end{aligned} \tag{5.4}$$

Example. Figure 5.2 shows the completing behaviour of a timer intermediate catching event (*Wait*) with dateTime configuration to ($2021-04-16\ T\ 00:00:00\ Z$). The left part of the figure shows that $Wait$ is enabled to complete: it has a token on its incoming edge ($e2$), and the global clock of the diagram meets the fixed date defined on it. The right part of the figure shows that $Wait$ completes by removing a token from its incoming edge ($e2$), producing a token on its outgoing sequence flow edge ($e3$).

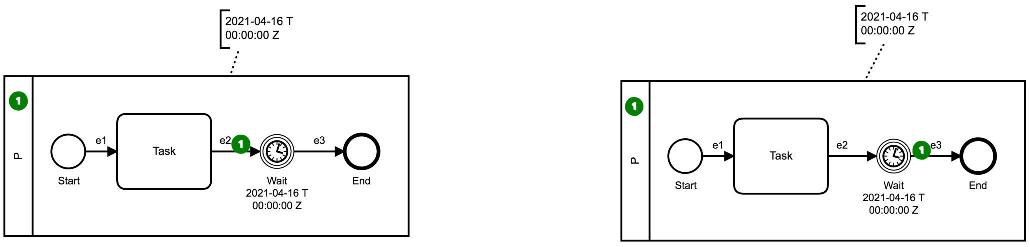


Figure 5.2: *Starting Behaviour of a Timer Intermediate Catch Event (Date). Before (left) and after (right) application of the Ct rule.*

The behaviour of a $TICE_p$ event is different due to the local clocks associated with it. A $TICE_p$ is defined by a completing predicate. It is enabled to complete if one of its incoming edges has a token and the time of its local clock has met its deadline. It completes by consuming a token from one of its marked incoming edges, resetting its local clock, and generating a token on its outgoing edges. Formally:

$$\begin{aligned} Ct(n) \overset{def}{\equiv}\ & (cat_N(n) = TICE_p) \wedge (ftime(n) \downharpoonright_{timeVal_P} = l_c(n)) \wedge (l'_c(n) = 0) \\ & \wedge\ \exists e \in intype(n, SF), (me(e) = 1) \wedge (me'(e) = me(e) - 1) \\ & \wedge\ \forall e' \in outtype(n, SF), (me'(e') = me(e') + 1) \\ & \wedge\ \triangle(\{e\} \cup outtype(n, SF)) \wedge \triangle_t(\{n\}) \wedge \Xi \end{aligned} \tag{5.5}$$

Example. Figure 5.3 shows the completing behaviour of a timer intermediate catching event ($Wait$) with a duration configuration set to $P10D$ (10 days). The left part of the figure shows that $Wait$ is enabled to complete: it has a token on its incoming edge ($e2$), and its clock meets the deadline of 10 days from its activation (864000). The right part of the figure shows that ($Wait$) completes by removing one token from its incoming edge ($e2$), producing a token on its outgoing sequence flow edge ($e3$), and resetting its local clock.

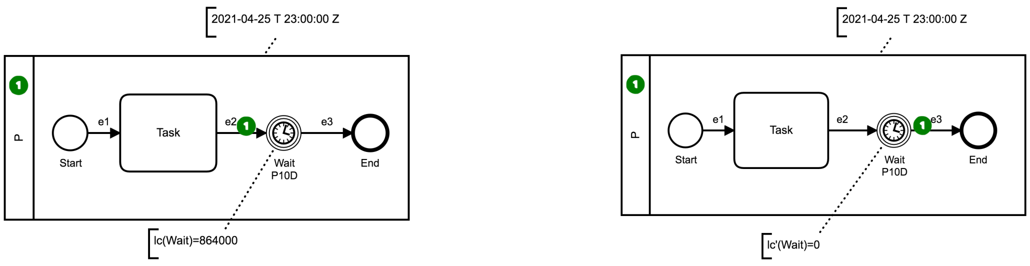


Figure 5.3: *Completing Behaviour of a Timer Intermediate Catch Event (Duration). Before (left) and after (right) application of the Ct rule.*

Timer Boundary Events (TBE). Are events attached to an activity. Such event can be either Interrupting ($TBE^{\oslash}$), *i.e.*, it interrupts the running of the activity it is attached to, or Non-Interrupting ($TBE^{\oplus}$). The start of activity with timer boundary events causes the activation of local clocks for the boundary events attached to it if they exist. Both types of the TBE event are defined by a starting predicate.

The Interrupting Timer Boundary Event ($TBE^{\oslash}$) acts as a deadline for an activity. If the activation token remains on the activity for more than a specific duration or fixed date, the timer event interrupts the activity to which it is attached. As we separate a $TBE^{\oslash}_d$ which is specified with a date and a $TBE^{\oslash}_p$ which is specified with a duration, we give their behaviour as follows.

An Interrupting Timer Boundary Events with Date ($TBE^{\oslash}_d$) is ready to start if the global clock of the model meets the fixed date and time defined on it and the activity attached to it is active (*i.e.*, owns a token). A $TBE^{\oslash}_d$ starts by cancelling the activity to which it is attached to if this activity is not a subProcess in its completing step (*i.e.*, it has at least one token on one of its elements other than its end events). An activity is cancelled by dropping all its tokens. The $TBE_d^{\oslash}$ then produces a token on each of its outgoing edges.

$$
\begin{array}{rl}
St(n) \overset{def}{\equiv} & (cat_N(n) = TBE_d^{\oslash}) \wedge \ (ftime(n) \downharpoonright_{timeVal_D} = g_c) \wedge isInterrupt(n) \\
& \wedge\ \exists act \in N, cat_N(act) \in \mathbb{A}, (act = attachedTo(n)) \wedge (mn(act) \geq 1) \\
& \wedge \left(\begin{array}{l}
\left(\begin{array}{l} cat_N(act) \notin SP \wedge (mn'(act) = 0) \\ \quad \wedge\ (\forall ee \in outtype(n, SF), (me'(ee) = me(ee) + 1)) \\ \quad \wedge\ \triangle (\{act\} \cup outtype(n, SF)) \wedge \triangle_t(\emptyset) \wedge \Xi \end{array} \right) \\
\vee \left(\begin{array}{l} cat_N(act) \in SP \wedge \neg mayComplete(act) \wedge mn'(act) = 0 \\ \quad \wedge\ (\forall nn \in R(act) \cap N, (mn'(nn) = 0) \\ \quad \wedge\ (\forall ee \in R(act) \cap E, (me'(ee) = 0) \\ \quad \wedge\ (\forall out \in outtype(n, SF), (me'(out) = me(out) + 1)) \\ \quad \wedge\ \triangle (\{act\} \cup R(act) \cup outtype(n, SF)) \wedge \triangle_t(\emptyset) \wedge \Xi) \end{array} \right)
\end{array} \right)
\end{array}
\tag{5.6}
$$

Example. Figure 5.4 shows the starting behaviour of an interrupting timer boundary event ($Interrupt$) with Date configuration to ($2021 - 04 - 16\ T\ 00 : 00 : 00\ Z$). The left part of the figure shows that $Interrupt$ is enabled to start: the activity it is attached to is active ($task1$), and the global clock of the diagram meets the fixed date defined on it ($2021 - 04 - 16\ T\ 00 : 00 : 00\ Z$). The right part of the figure shows that $Interrupt$ completes by dropping the token of the $task1$ and producing a token on its outgoing sequence flow edge ($e3$).

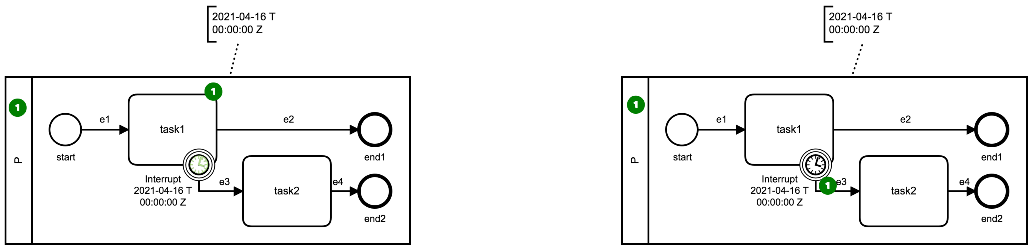


Figure 5.4: *Starting Behaviour of an Interrupting Timer Boundary Event (Date). Before (left) and after (right) application of the St rule.*

An Interrupting Timer Boundary Events with Duration ($TBE_p^{\oslash}$) is ready to start if its local clock meets its deadline and the activity attached to it is active (*i.e.*, owns a token). It starts by cancelling the activity to which it is attached to if this activity is not a subProcess in its completing step (*i.e.*, it has at least one token on one of its elements other than its end events). Again an activity is cancelled by dropping all its tokens. The $TBE_p^{\oslash}$ then resets its local clock and produces a token on each of its outgoing edges.

$$
\begin{array}{rl}
St(n) \overset{def}{\equiv} & (cat_N(n) = TBE_p^{\oslash}) \wedge \ (ftime(n) \downharpoonright_{timeVal_P} = l_c(n)) \wedge (l'_c(n) = 0)) \wedge isInterrupt(n) \\
& \wedge\ \exists act \in N, cat_N(act) \in \mathbb{A}, (act = attachedTo(n)) \wedge (mn(act) \geq 1) \\
& \wedge \left(\begin{array}{l}
\left(\begin{array}{l} cat_N(act) \notin SP \wedge (mn'(act) = 0) \\ \quad \wedge(\forall ee \in outtype(n, SF), (me'(ee) = me(ee) + 1)) \\ \quad \wedge\ \triangle (\{act\} \cup outtype(n, SF)) \wedge \triangle_t(\{n\}) \wedge \Xi \end{array} \right) \\
\vee \left(\begin{array}{l} cat_N(act) \in SP \wedge \neg mayComplete(act) \wedge mn'(act) = 0 \\ \quad \wedge\ (\forall nn \in R(act) \cap N, (mn'(nn) = 0) \\ \quad \wedge\ (\forall ee \in R(act) \cap E, (me'(ee) = 0) \\ \quad \wedge\ (\forall out \in outtype(n, SF), (me'(out) = me(out) + 1)) \\ \quad \wedge \triangle (\{act\} \cup R(act) \cup outtype(n, SF)) \wedge \triangle_t(\{n\}) \wedge \Xi) \end{array} \right)
\end{array} \right)
\end{array}
\tag{5.7}
$$

Example. Figure 5.5 shows the starting behaviour of an interrupting timer boundary event ($Interrupt$) with a duration configuration of $P10D$ (10 days). The left part of the figure shows that $Interrupt$ is enabled to start: the activity it is attached to is active ($task1$), and its local clock meets the deadline of 10 days from its activation (864000). The right part of the figure shows that ($Interrupt$) starts by dropping the token of the ($task1$), producing a token on its outgoing sequence flow edge ($e3$), and resetting its local clock.

Like $TBE^{\oslash}$ events, the Non-Interrupting Timer Boundary Events ($TBE^{\oplus}$) can be configured with a

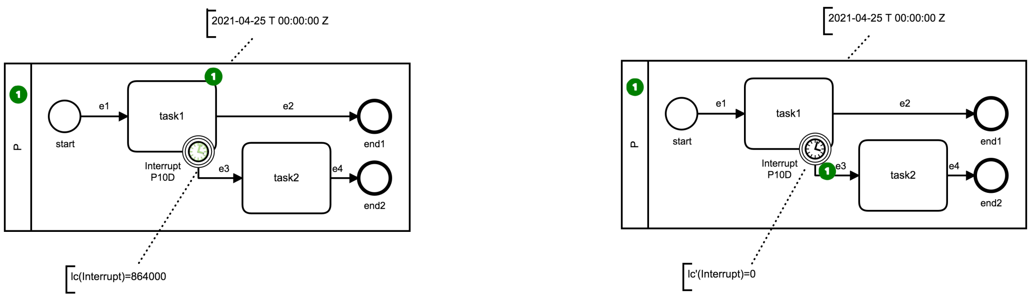


Figure 5.5: *Starting Behaviour of an Interrupting Timer Boundary Event (Duration). Before (left) and after (right) application of the St rule.*

date ($TBE_d^{\oplus}$), a duration ($TBE_p^{\oplus}$), but also a time cycle ($TBE_c^{\oplus}$). We give their behaviour separately as follows. Formally:

The Non-Interrupting Timer Boundary Events with Date ($TBE_d^{\oplus}$) define the same behaviour as $TBE_d^{\oslash}$ without cancelling the activity that they are attached to it.

$$
\begin{aligned}
St(n) \overset{def}{\equiv}\ & (cat_N(n) = TBE_d^{\oplus}) \wedge (ftime(n) \downharpoonright_{timeVal_D} = g_c) \wedge \neg isInterrupt(n) \\
& \wedge\ \exists act \in N, cat_N(act) \in \mathbb{A}, (act = attachedTo(n)) \wedge (mn(act) \geq 1) \\
& \wedge\ (\forall out \in outtype(n, SF), (me'(out) = me(out) + 1)) \\
& \wedge\ \triangle\ (\{act\} \cup R(act) \cup outtype(n, SF)) \wedge \triangle_t(\emptyset) \wedge \Xi
\end{aligned} \tag{5.8}
$$

Example. Figure 5.6 shows the starting behaviour of a non-interrupting timer boundary event ($Non-Interrupt$) with dateTime configuration to ($2021-04-16\ T\ 00:00:00\ Z$). The left part of the figure shows that $Non-Interrupt$ is enabled to start: the activity it is attached to is active ($task1$), and the global clock of the diagram meets the fixed date defined on it ($2021-04-16\ T\ 00:00:00\ Z$). The right part of the figure shows that $Non-Interrupt$ completes by producing a token on its outgoing sequence flow edge ($e3$).

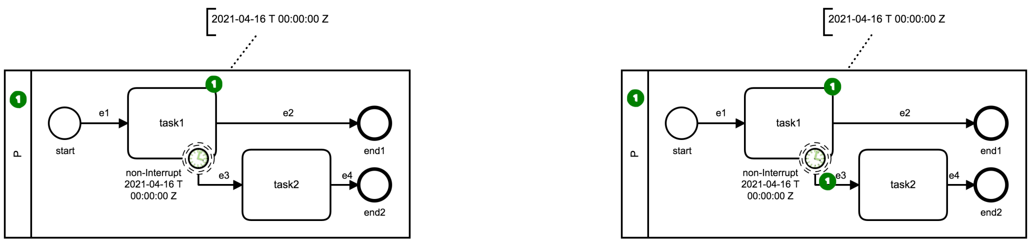


Figure 5.6: *Starting Behaviour of a Non-Interrupting Timer Boundary Event (Duration). Before (left) and After (right) application of the St rule.*

The Non-Interrupting Timer Boundary Events with Duration ($TBE_p^{\oplus}$) define the same behaviour as $TBE_p^{\oslash}$, without cancelling the activity that they are attached to it. Formally:

$$
\begin{aligned}
St(n) \overset{def}{\equiv}\ & (cat_N(n) = TBE_p^{\oplus}) \wedge (\neg isInterrupt(n)) \\
& \wedge\ (\exists act \in N, cat_N(act) \in \mathbb{A}, (act = attachedTo(n)) \wedge (mn(act) \geq 1) \\
& \wedge\ (ftime(n) \downharpoonright_{timeVal_P} = l_c(n)) \wedge (l_c'(n) = 0) \\
& \wedge\ (\forall out \in outtype(n, SF), (me'(out) = me(out) + 1)) \\
& \wedge\ \triangle\ (\{act\} \cup R(act) \cup outtype(n, SF)) \wedge \triangle_t(\{n\}) \wedge \Xi)
\end{aligned} \tag{5.9}
$$

Example. Figure 5.7 shows the starting behaviour of a non-interrupting timer boundary event ($Non-Interrupt$) with duration configuration to $P10D$ (10 days). The left part of the figure shows that $non-Interrupt$ is enabled to start: the activity that is attached to it is active ($task1$), and its local clock meets the deadline of 10 days from its activation (864000). The right part of the figure shows that ($Non-Interrupt$) starts by producing a token on its outgoing sequence flow edge ($e3$) and resetting its local clock.

The Non-Interrupting Timer Boundary Events with cycle configuration ($TBE_c^{\oplus}$) might be triggered multiple times while the activity it is attached to is active. The number of cycles can either be fixed

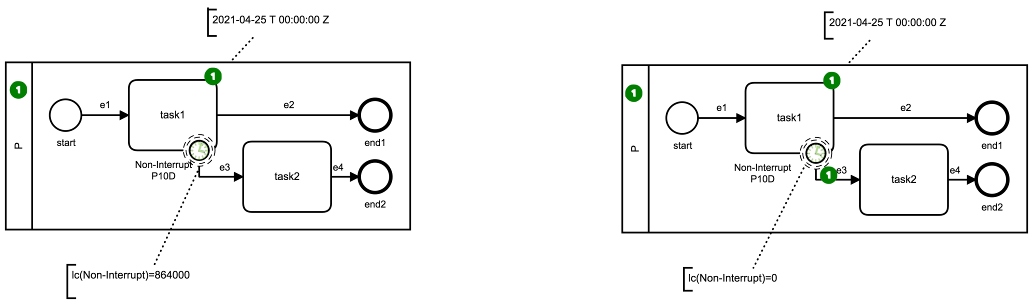


Figure 5.7: *Starting Behaviour of a Non-Interrupting Timer Boundary Event (Duration). Before (left) and after (right) application of the St rule.*

or unbounded. Therefore, the time cycle definition associated with a timer event may have different configurations:

(i) A Non-Interrupting Timer Boundary Event defines a number of recurrences for the timer event separated by a period and where the first trigger is relatively done to a fixed start date ($TBE^{\oplus}_{c(start)}$). The behaviour of $TBE^{\oplus}_{c(start)}$ is defined by starting and completing predicates as follows.

The $TBE^{\oplus}_{c(start)}$ is ready to start if the activity attached to it is active, the global clock of the model meets the fixed date and time defined on it, and its local clock is inactive. It starts by activating its local clock, generating a token on its outgoing edges, and decreasing the recurrence number by one if bounded to a number n. Formally:

$$\begin{aligned}
St(n) \overset{def}{\equiv}\ & (cat_N(n) = TBE^{\oplus}_{c(start)}) \wedge (\neg isInterrupt(n)) \wedge (rec(n) = ftime(n)\downharpoonright_{timeVal_R})\\
& \wedge\ (\exists act \in N, cat_N(act) \in \mathbb{A}, (act = attachedTo(n)) \wedge (mn(act) \geq 1)\\
& \wedge \begin{pmatrix} ((rec(n) = \iota) \wedge (rec'(n) = rec(n)))\\ \vee((rec(n) \in \mathbb{N}) \wedge (rec'(n) = rec(n) - 1)) \end{pmatrix}\\
& \wedge\ (ftime(n)\downharpoonright_{timeVal_D} = g_c) \wedge (l_c(n) = 0) \wedge (l'_c(n) = 1)\\
& \wedge\ (\forall ee \in outtype(n, SF), (me'(ee) = me(ee) + 1))\\
& \wedge\ \triangle\,(outtype(n, SF)) \wedge \triangle_t(\{n\}) \wedge \Xi)
\end{aligned} \tag{5.10}$$

Example. Figure 5.8 shows the starting behaviour of a bounded Non-interrupting timer boundary event ($non - InterruptC$) with a two cycles configuration that starts at $2021 - 04 - 16T00 : 00 : 00$ and re-executes for each period of 10 days ($P10D$). The left part of the figure shows that $Non - InterruptC$ is enabled to start: the activity that it is attached to it is active ($task1$), the global clock meets the starting date, its local clock is inactive, and the recurrence number is as defined equal to 2. The right part of the figure shows that $non - Interrupt$ starts by producing a token on its outgoing sequence flow edge ($e3$), activating its local clock, and decrementing the recurrence number by one.

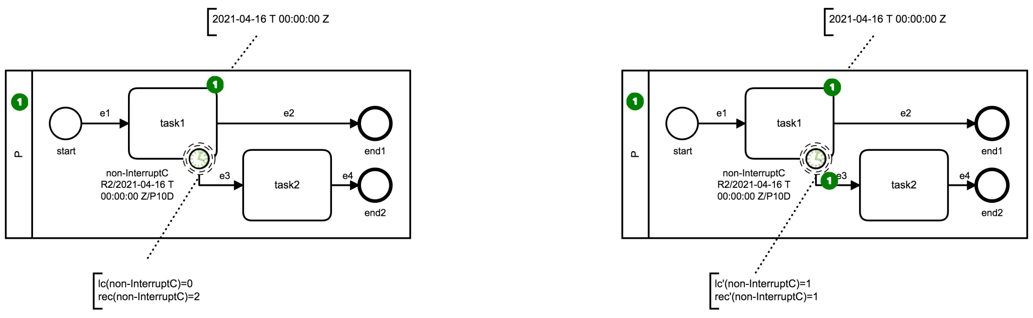


Figure 5.8: *Starting Behaviour of a Non-Interrupting Timer Boundary Event (Cycle) with a Fixed Start TimeDate and Duration. Before (left) and after (right) the first application of the St rule.*

A $TBE^{\oplus}_{c(start)}$ event is ready to complete if the activity that is attached to it is active, the local clock of the node reaches its duration, and the number of recurrences has not reached 0 if it is bounded. It completes by re-activating its local clock, generating a token on its outgoing edges, and decreasing the recurrence number by one if bounded to a number n.

$$
\begin{aligned}
Ct(n) \overset{def}{\equiv}\ & (cat_N(n) = TBE^{\oplus}_{c(start)}) \wedge (\neg isInterrupt(n)) \\
& \wedge\ (\exists act \in N, act \in \mathbb{A}, (act = attachedTo(n)) \wedge (mn(act) \geq 1) \\
& \wedge\ (ftime(n) \downharpoonleft_{timeVal_P} = l_c(n)) \ \wedge\ (l'_c(n) = 1) \\
& \wedge \left(\begin{array}{l} ((rec(n) = \iota) \wedge (rec'(n) = rec(n))) \\ \vee((rec(n) \in \mathbb{N}) \wedge (rec(n) \neq 0) \wedge (rec'(n) = rec(n) - 1)) \end{array} \right) \\
& \wedge\ (\forall ee \in outtype(n, SF), (me'(ee) = me(ee) + 1)) \\
& \wedge\ \triangle\,(outtype(n, SF)) \wedge \triangle_t(\{n\}) \wedge \Xi)
\end{aligned}
\tag{5.11}
$$

Example. Figure 5.9 shows the completing behaviour of a bounded non-interrupting timer boundary event $(non - InterruptC)$ with a two cycles configuration that starts at $2021 - 04 - 16$ T $00:00:00$ and re-executes for each period of 10 days $(P10D)$. The left part of the figure shows that $non - InterruptC$ is enabled to complete: the activity it is attached to is active $(task1)$, its local clock meets the deadline of 10 days from its activation (864000), and the recurrence number is greater than 0. The right part of the figure shows that $(non - InterruptC)$ completes by producing a token on its outgoing sequence flow edge $(e3)$ and re-activating its local clock.

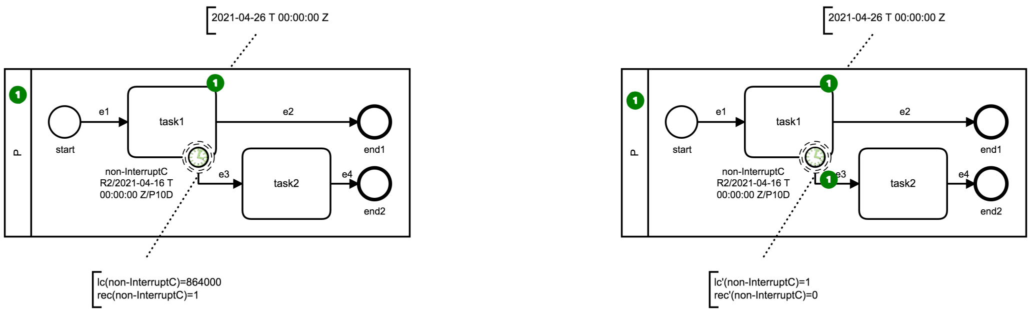


Figure 5.9: *Competing Behaviour of a non-Interrupting Timer Boundary Event (Cycle) with a Fixed Start TimeDate and Duration. Before (left) and after (right) application of the Ct rule.*

(ii) A Non-Interrupting Timer Boundary Event defines a number of recurrences for the timer event triggers separated by a period and where the last trigger is done before the fixed end date $(TBE^{\oplus}_{c(end)})$. The behaviour of $TBE^{\oplus}_{c(end)}$ is defined only by a starting predicate. It is ready to start if the activity attached to it is active, the node's local clock reaches its duration, the global clock does not meet the fixed date and time yet, and the number of recurrences is not equal to 0 if it is bounded. It starts by re-activating its local clock, generating a token on its outgoing edges, and decreasing the recurrence number by one if bounded to a number n. Formally:

$$
\begin{aligned}
St(n) \overset{def}{\equiv}\ & (cat_N(n) = TBE^{\oplus}_{c(end)}) \wedge (\neg isInterrupt(n)) \wedge\ (ftime(n) \downharpoonleft_{timeVal_D} \neq g_c) \\
& \wedge\ (\exists act \in N, cat_N(act) \in \mathbb{A}, (act = attachedTo(n)) \wedge (mn(act) \geq 1) \\
& \wedge(ftime(n) \downharpoonleft_{timeVal_P} = l_c(n)) \wedge (l'_c(n) = 1) \\
& \wedge \left(\begin{array}{l} ((rec(n) = \iota) \wedge (rec'(n) = rec(n))) \\ \vee((rec(n) \in \mathbb{N}) \wedge (rec(n) \neq 0) \wedge (rec'(n) = rec(n) - 1)) \end{array} \right) \\
& \wedge\ (\forall ee \in outtype(n, SF), (me'(ee) = me(ee) + 1)) \\
& \wedge\ \triangle\,(outtype(n, SF)) \wedge \triangle_t(\{n\}) \wedge \Xi)
\end{aligned}
\tag{5.12}
$$

Example. Figure 5.10 shows the starting behaviour of a bounded non-interrupting timer boundary event $(non - InterruptC)$ with a two cycles configuration that may re-execute twice for each period of 10 days $(P10D)$ before date $2021 - 04 - 30$ T $00:00:00$. The left part of the figure shows that $non - InterruptC$ is enabled to start: the activity it is attached to is active $(task1)$, its local clock meets the deadline of 10 days from its activation (864000), the global clock has not reached yet $(2021 - 04 - 30$ T $00:00:00)$, and the recurrence number is greater than 0. The right part of the figure shows that $(non - InterruptC)$ starts by producing a token on its outgoing sequence flow edge $(e3)$ and re-activating its local clock.

(iii) A Non-Interrupting Timer Boundary Event defines a number of recurrences for the timer event triggers separated by a period $(TBE^{\oplus}_{c(p)})$. Only a starting predicate defines the behaviour of $TBE^{\oplus}_{c(p)}$. It is ready to start if the activity attached to it is active, the node's local clock reaches its duration, and the number of recurrences is not equal to 0 if it is bounded. It starts by re-activating its local clock,

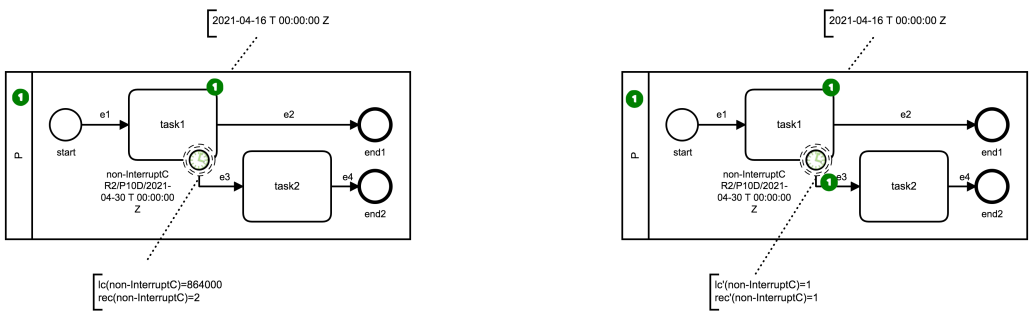


Figure 5.10: *Staring behaviour of a Non-Interrupting Timer Boundary Event (Cycle) with a Duration and a Fixed Last TimeDate. Before (left) and after (right) application of the St rule.*

generating a token on its outgoing edges, and decreasing the recurrence number by one if bounded to a number n. Formally:

$$
\begin{aligned}
St(n) \overset{def}{\equiv}\ & (cat_N(n) = TBE^{\oplus}_{c(p)}) \\
& \wedge\ (\exists act \in N, (cat_N(n) = \mathbb{A}) \wedge (act = attachedTo(n)) \wedge (mn(act) \geq 1) \wedge (\neg isInterrupt(n)) \\
& \wedge\ (ftime(n) \downharpoonright_{timeVal_P} = l_c(n)) \wedge (rec'(n) = rec(n) - 1) \wedge (l'_c(n) = 1) \\
& \wedge\ (\forall ee \in outtype(n, SF), (me'(ee) = me(ee) + 1)) \\
& \wedge\ \triangle (outtype(n, SF)) \wedge \triangle_t(\{n\}) \wedge \Xi)
\end{aligned} \tag{5.13}
$$

Example. Figure 5.11 shows the starting behaviour of a bounded non-interrupting timer boundary event 2800 $(non - InterruptC)$ with a two cycles configuration that may re-execute twice for each period of 10 days $(P10D)$. The left part of the figure shows that $non - InterruptC$ is enabled to start: the activity it is attached to is active $(task1)$, its local clock meets the deadline of 10 days from its activation (864000), and the recurrence number is greater than 0. The right part of the figure shows that $(non - InterruptC)$ starts by producing a token on its outgoing sequence flow edge $(e3)$, re-activating its local clock.

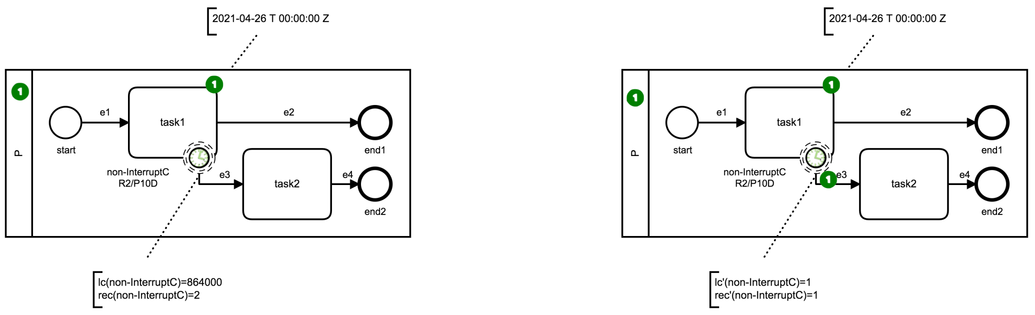


Figure 5.11: *Staring Behaviour of a Non-Interrupting Timer Boundary Event (Cycle) with a Duration. Before (left) and after (right) application of the St rule.*

2805 The provided semantics for the timer events also requires an extended semantics behaviour for some other non-temporal elements as follows:

Event-Based Gateway. With the introduction of the timed semantics, the semantics of the event-based gateways must be changed. In the standard, the execution semantics of an event-based gateway is defined as a branching point where exactly one of its outgoing edges is activated, depending on which 2810 event is triggered. Then, the path to that event will be used (a token will be sent down the outgoing sequence flows of the event) [3].

In a BPMN model, an event-based gateway is followed by a receive task (RT) or an intermediate catching message event $(CMIE)$, combined with a timer intermediate catch event $(TICE)$. The activation of one of the outgoing edges depends on the enabling of these elements (*i.e.*, the reception of a 2815 message, or a specific time event being triggered). So, to handle time, we adapt the semantics of an event-based gateway as follows.

An Event-based gateway (EB) is defined only by a completing predicate. It is ready to complete if one of its incoming edges has a token, and one of its target events is enabled (*i.e.*, the target of an outgoing edge is an RT or a $CMIE$ that has an offer on one of its incoming message edges, or the target of an outgoing edge is an intermediate timer event with a local clock that meets a deadline or when the global clock reaches the event timeDate). The EB completes by consuming the token from one of its incoming edges and producing a token on the outgoing edge on which the event is enabled. Formally:

$$\begin{aligned}
Ct(n) \overset{def}{\equiv}\ & (cat_N(n) = EB)\\
& \wedge\ \exists\ e \in intype(n, SF), (me(e) \geq 1) \wedge (me'(e) = me(e) - 1)\\
& \wedge\ \exists e' \in outtype(n, SF),\ (me'(e') = me(e') + 1)\\
& \qquad \wedge \begin{pmatrix} (cat_N(tgt(e')) \in \{RT, CMIE\} \wedge \exists mf \in intype(tgt(e'), MF), (me(mf) \geq 1)) \\ \vee\ (cat_N(tgt(e')) = TICE_p \wedge (l_c(tgt(e')) \geq ftime\downharpoonright_{timeVal_P} (tgt(e')))) \\ \vee\ (cat_N(tgt(e')) = TICE_d \wedge (g_c \geq ftime\downharpoonright_{timeVal_D} (tgt(e')))) \end{pmatrix}\\
& \wedge \triangle(\{e, e'\}) \wedge \triangle_t(\emptyset) \wedge \Xi
\end{aligned} \tag{5.14}$$

Example. Figure 5.12 shows a collaboration diagram between two active processes (P, Q). The event-based gateway (EB) in the process P is followed by two message catch events ($FirstMsg$, $SecondMsg$) and a timer catch event ($Wait$) with a 10 days ($P10D$) duration configuration. The left part of the figure shows that EB is enabled to complete: its incoming edge ($e2$) has a token, and the target timer event of its outgoing edge ($e7$) is enabled to start (its local clock meets the deadline of 10 days from its activation (864000)). The right part of the figure shows that the gateway completes by consuming the token from its incoming edge ($e2$) and producing a token on its outgoing sequence flow edge ($e7$).

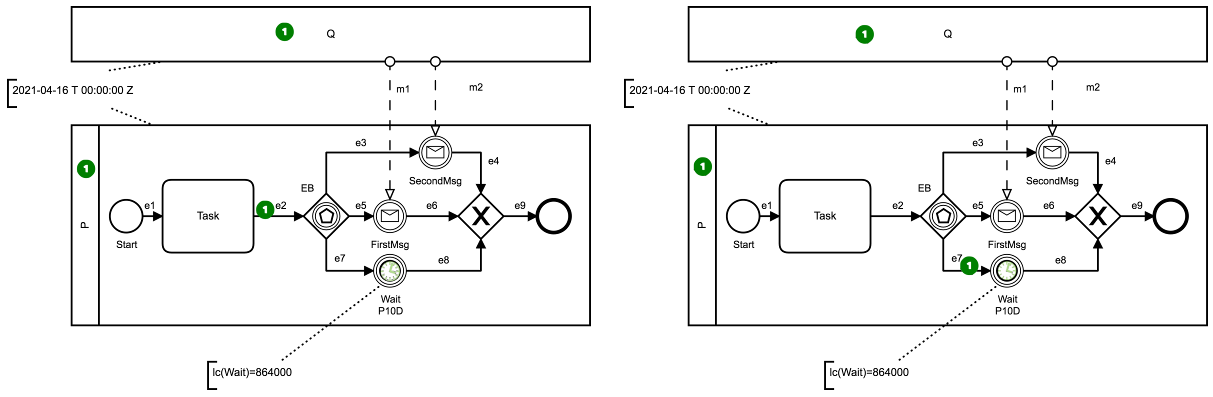


Figure 5.12: *Completing Behaviour of an Event-Based Gateway: the Timer is Ready to Fire. Before (left) and after (right) application of the Ct rule.*

Activity. As we saw before, the main working units in a process are the activities. We made the distinction between a composite activity, or *SubProcess* (SP), and an atomic activity, or *Task (T)*. The latter can be an Abstract (AT), a Send (ST), or a Receive (RT) task. All activities have the same basic behaviour. They start by moving a token from an incoming edge to themselves. When the activity is associated with a timer boundary event that requires a local clock ($TBE_p^{\oplus}$,$TBE_{c(p)}^{\oplus}$, $TBE_{c(end)}^{\oplus}$, or $TBE_p^{\oslash}$), there are some additional constraints w.r.t the ones described in 4.4.0.2. When an activity starts, it starts all inactive local clocks of its attached boundary events if they exist, and when it completes, it deactivates them. We give the starting and completing formula for all supported task types and the subprocess in the following. To simplify the formula, we define a specific set of timer nodes, called $TimerP$, that groups all-timer nodes configured with a duration as:

$$TimerP \overset{def}{\equiv} \{t \in N \mid cat_N(t) \in \{TBE_p^{\oplus}, TBE_{c(p)}^{\oplus}, TBE_{c(end)}^{\oplus}, TBE_p^{\oslash}\}\}$$

- **Abstract Task**

$$\begin{aligned} St(n) \overset{def}{\equiv}\ & (cat_N(n) = AT) \wedge\ (\exists e \in intype(n, SF), (me(e) \geq 1) \wedge (me'(e) = me(e) - 1) \\ & \wedge\ (mn(n) = 0) \wedge (mn'(n) = mn(n) + 1) \\ & \wedge\ (\forall te \in N, (cat_N(te) \in TimerP) \wedge (n = attachedTo(te)) \wedge\ (l_c(te) = 0) \Rightarrow (l'_c(te) = 1)) \\ & \wedge\ \triangle_t\ (\{te \in N, cat_N(te) \in TimerP \mid (n = attachedTo(te))\}) \wedge\ \triangle(\{n, e\})) \wedge \Xi \end{aligned} \tag{5.15}$$

$$\begin{aligned} Ct(n) \overset{def}{\equiv}\ & (cat_N(n) = AT) \wedge (mn(n) \geq 1) \wedge (mn'(n) = mn(n) - 1) \\ & \wedge\ (\forall e \in outtype(n, SF), (me'(e) = me(e) + 1)) \\ & \wedge\ (\forall te \in N^{TimerP}, (n = attachedTo(te)) \Rightarrow (l'_c(n) = 0)) \\ & \wedge\ \triangle_t\ (\{te \in N, cat_N(te) \in TimerP \mid (n = attachedTo(te))\}) \\ & \wedge\ \triangle\ (\{n\} \cup outtype(n, SF)) \wedge \Xi \end{aligned} \tag{5.16}$$

- **Send Task**

$$\begin{aligned} St(n) \overset{def}{\equiv}\ & (cat_N(n) = ST) \wedge (\exists e \in intype(n, T_{SP}), (me(e) \geq 1) \wedge (me'(e) = me(e) - 1) \\ & \wedge\ (mn(n) = 0) \wedge (mn'(n) = mn(n) + 1) \\ & \wedge\ (\forall te \in N, (cat_N(te) \in TimerP) \wedge (n = attachedTo(te)) \wedge\ (l_c(te) = 0) \Rightarrow (l'_c(te) = 1)) \\ & \wedge \triangle_t\ (\{te \in N, cat_N(te) \in TimerP \mid (n = attachedTo(te))\})\ \triangle\ (\{n, e\}) \wedge \Xi) \end{aligned} \tag{5.17}$$

$$\begin{aligned} Ct(n) \overset{def}{\equiv}\ & (cat_N(n) = ST) \wedge (mn(n) = 1) \wedge (mn'(n) = mn(n) - 1) \\ & \wedge\ \forall e \in outtype(n, SF), (me'(e) = me(e) + 1) \\ & \wedge\ (\forall te \in N^{TimerP}, (n = attachedTo(te)) \Rightarrow (l'_c(n) = 0)) \\ & \wedge\ (\exists ee \in outtype(n, MF), (me'(ee) = me(ee) + 1) \\ & \quad \wedge\ send(procOf(n), procOf(target(ee)), msg_t(ee)) \\ & \quad \wedge\ \triangle\ (\{n, ee\} \cup outtype(n, SF))) \\ & \wedge\ \triangle_t\ (\{te \in N, cat_N(te) \in TimerP \mid (n = attachedTo(te))\}) \end{aligned} \tag{5.18}$$

- **Receive Task**

$$\begin{aligned} St(n) \overset{def}{\equiv}\ & (cat_N(n) = RT) \wedge (\exists e \in intype(n, SF), (me(e) \geq 1) \wedge (me'(e) = me(e) - 1) \\ & \quad \wedge\ (mn(n) = 0) \wedge (mn'(n) = mn(n) + 1) \\ & \quad \wedge \triangle\ (\{n, e\}) \wedge \Xi) \\ & \wedge\ (\forall te \in N, (cat_N(te) \in TimerP) \wedge (n = attachedTo(te)) \wedge\ (l_c(te) = 0) \\ & \quad \Rightarrow (l'_c(te) = 1)) \\ & \wedge \triangle_t\ (\{te \in N, cat_N(te) \in TimerP \mid (n = attachedTo(te))\})\ \triangle\ (\{n, e\}) \end{aligned} \tag{5.19}$$

$$\begin{aligned} Ct(n) \overset{def}{\equiv}\ & (cat_N(n) = RT) \wedge (mn(n) = 1) \wedge (mn'(n) = mn(n) - 1) \\ & \wedge\ \forall e \in outtype(n, SF), (me'(e) = me(e) + 1) \\ & \wedge\ (\forall te \in N^{TimerP}, (n = attachedTo(te)) \Rightarrow (l'_c(n) = 0)) \\ & \wedge\ (\exists ee \in intype(n, MF), (me(ee) \geq 1) \wedge (me'(ee) = me(ee) - 1) \\ & \quad \wedge\ receive(procOf(source(ee)), procOf(n), msg_t(ee)) \\ & \quad \wedge \triangle\ (\{n, ee\} \cup outtype(n, SF))) \\ & \wedge\ \triangle_t\ (\{te \in N, cat_N(te) \in TimerP \mid (n = attachedTo(te))\}) \end{aligned} \tag{5.20}$$

- **SubProcess**

$$\begin{aligned} St(n) \overset{def}{\equiv}\ & (cat_N(n) = SP) \\ & \wedge\ (\exists e \in intype(n, SF), (me(e) \geq 1) \wedge (me'(e) = me(e) - 1) \wedge (mn'(n) = mn(n) + 1) \\ & \wedge\ (\forall n_{se} \in R(n), (cat_N(n_{se}) \in NSE) \wedge (mn'(n_{se}) = mn(n_{se}) + 1)) \\ & \wedge\ (\forall te \in N, (cat_N(te) \in TimerP) \wedge (n = attachedTo(te)) \Rightarrow (l'_c(n) = 1)) \\ & \wedge\ \triangle\ (\{e, n\} \cup (\{n_{se} \in R(n), cat_N(n_{se}) \in NSE\}))) \\ & \wedge\ \triangle_t\ (\{te \in N, cat_N(te) \in TimerP \mid (n = attachedTo(te))\}) \wedge \Xi \end{aligned} \tag{5.21}$$

$$
\begin{aligned}
Ct(n) \overset{def}{\equiv}\ & (cat_N(n) = SP) \wedge (mn(n) \geq 1) \wedge (mn'(n) = mn(n) - 1) \\
& \wedge\ (\forall e \in R(n) \cap E, (me(e) = 0)) \\
& \wedge\ (\exists n_{ee} \in R(n), (cat_N() = EE) \wedge (mn(n_{ee}) \geq 1)) \\
& \wedge\ (\forall nn \in R(n) \cap N, (mn(nn) \geq 1 \Rightarrow cat_N(nn) \in EE)) \\
& \wedge\ (\forall nn \in R(n), cat_N(nn) \in EE) \wedge (mn'(nn) = 0)) \\
& \wedge\ (\forall e \in outtype(n, SF), (me'(e) = me(e) + 1)) \\
& \wedge\ (\forall te \in N, cat_N(te) \in TimerP, (n = attachedTo(te)) \Rightarrow (l'_c(te) = 0)) \\
& \wedge\ \triangle(\{n\} \cup \{nn \in R(n), cat_N(n) \in EE\} \cup outtype(n, SF)) \\
& \wedge\ \triangle_t(\{te \in N, cat_N(te) \in TimerP \mid (n = attachedTo(te))\}) \wedge \Xi
\end{aligned}
\tag{5.22}
$$

Example. Figure 5.13 shows the starting behaviour of a task ($task1$) and a subProcess (SP) with attached interrupting timer boundary events with a duration configuration of 10 days ($P10D$), called ($timeOut_1$) and ($timeOut_2$) respectively. The left part of the figure shows that $task1$ and SP are
2835 enabled to start: they are inactive, and there is a token on their incoming edges. The $task1$ node starts by consuming a token from its incoming edge, generating one on itself, and activating the local clock of its attached timer node $timeOut_1$. The SP node starts by consuming a token from its incoming edge, generating one on itself and its start event $start_{SP}$, and activating the local clock of its attached timer node $timeOut_2$. Figure 5.14 shows the completing behaviour of task $task1$ and subProcess SP. The left
2840 part of the figure shows that $task1$ and SP are enabled to complete: the $task1$ is active, SP is active, and its token has reached its end event. They are not interrupted yet. They complete by consuming their tokens, deactivating their attached local clocks the $timeOut_1$ and the $timeOut_2$, and generating a token on their outgoing edges $e2$, $e8$ respectively.

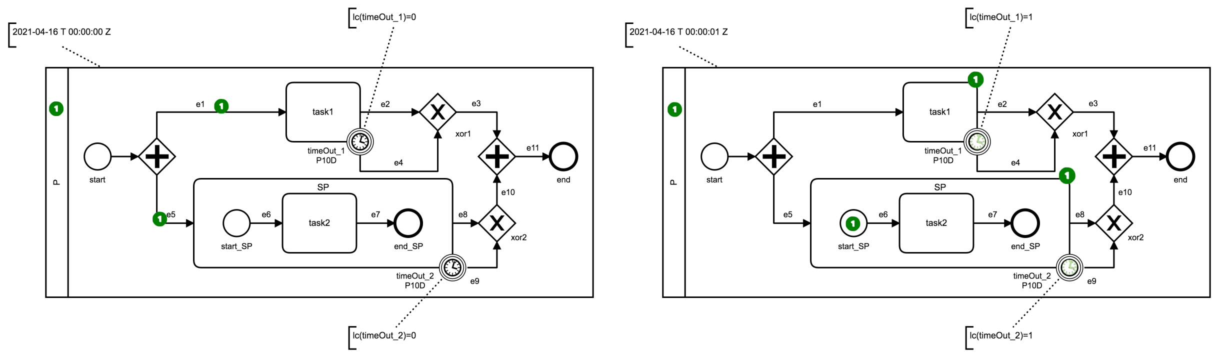


Figure 5.13: *Staring Behaviour of an Activity with Interrupting Boundary Timer Event Configured with Duration: Before (left) $task1$ and SP are ready to start and after (right) application of the StActivity rule twice (once for $task1$, once for SP.*

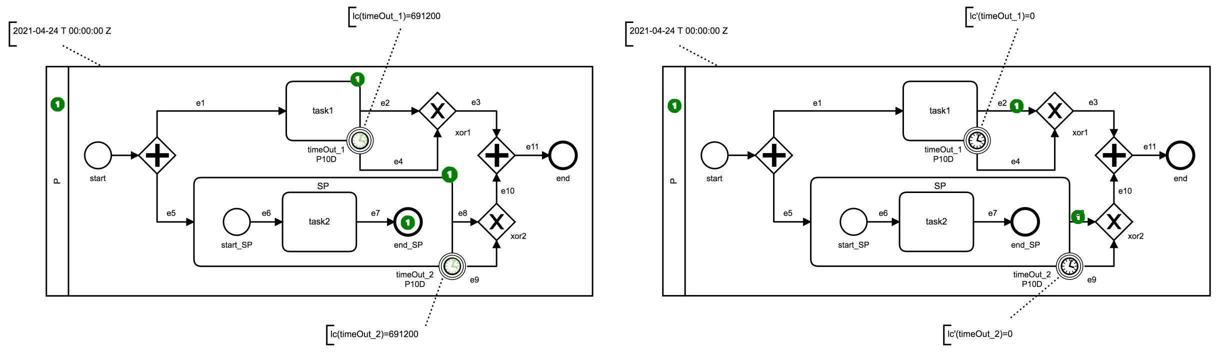


Figure 5.14: *Completing Behaviour of an Activity with Interrupting Boundary Timer Event Configured with Duration: Before (left) $task1$ and SP are ready to complete and after (right) application of the Ct rule (twice again).*

Note that the semantics of the elements that are not mentioned here (NSE, MSE, XOR, AND,
2845 OR, NEE, TEE) have the same behaviour defined before in Chapter 4.

5.3.2 Transition Relation and Executions

We can now express the complete transition relation (successor relation between states) with the previously defined predicates. For simplifying the transition definition, let us consider a subset of timer nodes, called T, that groups all the timer nodes in a given graph model. Formally we define T as follows:

$$T \overset{def}{\equiv} \{t \in N \mid cat_N(t) \in Timer\}$$

Let us consider a subset of timer nodes, called S, that groups the nodes that satisfy any of the following conditions:

- a starting timer node that has a token and such as the global time date of the system has not reached the fixed time date;
- an intermediate timer node that has an inactive local clock and has a marked incoming edge, or that follows an event-based gateway with a marked incoming edge;
- a boundary timer node attached to an active activity with a local clock that is not active;
- an active timer node, whose local clock is greater than 0 and has not reached the timing limits.

Formally, we define S as follows:

$$\begin{aligned}
S \overset{def}{\equiv}\ & \{n \in N, cat_N(n) = TSE \mid (ftime(n) \lfloor_{timeVal} < g_c) \wedge (mn(n) = 1)\} \\
\cup\ & \{n \in N, cat_N(n) = TICE \mid \exists e \in intype(n, SF), (l_c(n) = 0) \wedge (me(e) > 0)\} \\
\cup\ & \{n \in N, cat_N(n) = TICE \mid \exists e \in intype(src(intype(n, SF)), SF), (l_c(n) = 0) \wedge\ (me(e) > 0)\} \\
\cup\ & \{n \in N, cat_N(n) = TBE \mid (l_c(n) = 0) \wedge (mn(attachedTo(n)) > 0)\} \\
\cup\ & \{n \in T \mid ftime(n) \lfloor_{timeVal} > l_c(n) > 0\}
\end{aligned}$$

Let Y be the subset of timer nodes in the BPMN graph that are ready to fire:

$$Y \overset{def}{\equiv} \{y \in T \mid l_c(y) \geq ftime(n) \lfloor_{timeVal} (y) \vee g_c = ftime(n) \lfloor_{timeVal} (y)\}$$

To facilitate the reading of the transition relation, we define the following predicates:

- *step* defines a step of execution for a given node:

$$step(n) \overset{def}{\equiv} St(n) \vee Ct(n)$$

- $fztime$ denotes time equality for the local clock of all timer nodes given as parameter:

$$fztime(Z) \overset{def}{\equiv} \forall z \in Z, l'_c(z) = l_c(z)$$

The transition relation will be defined according to the following rules.

- If there are timer nodes that are in an active state and are enabled to be complete, they have a priority:
  - They will run one by one. Each node runs according to its execution rule;
  - During their execution, neither the global clock nor their local clocks can be incremented.

  If no timer node is ready to complete, and there is a non-timer node that may start or complete, then:

  - Any enabled node may be executed to start or complete;
  - The time of all inactive timer nodes is frozen;
  - The local time of all active timer nodes advances;
  - The global clock of the model advances.
- If no node may be executed and there is a timer node still active and not ready to complete, time advances only.

Thus, the transition relation distinguishes two cases.

If at least a timer is ready to fire ($Y \neq \emptyset$), then a timer fires (it does a step), or an event-based gateway that precedes a firable timer does a step. Time does not advance, and other timers with the same expiration time can then fire, *e.g.*, the two steps for task1 and SP in Figure 5.13. If no timer is ready to fire, all timers increase (run) and non-deterministically a step can occur ($\exists n, step(n)$) or no step is done ($\triangle(\emptyset) \wedge \Xi$). Thus, the transition relation distinguishes two cases. If at least a timer is ready to fire ($Y \neq \emptyset$), then a timer fires (it does a step) or an event-based gateway that precedes a firable timer does a step. Time does not advance, and other timers with the same expiration time can then fire. If no timer is ready to fire, all timers increase (run) and non-deterministically, a step can occur ($\exists n, step(n)$) or no step is done ($\triangle(\emptyset) \wedge \Xi$).

Definition 5.3.3 (Transition Relation)**.** The transition relation is a successor relation between states. It specifies that either a node makes a step (start or complete) or time advances, but only if no timer node is ready to complete. Let s and s' be two states. We say that s' is a successor of s, iff the predicate $Next(s, s')$ (See equation. 5.23) holds.

$$\begin{aligned} Next(s,s') \overset{def}{\equiv} (Y \neq \emptyset \wedge & \left(\begin{array}{c} (\exists n \in Y : step(n) \wedge fztime(T \setminus \{n\})) \\ \vee \left(\begin{array}{c} \exists n \in N, cat_N(n) = EB, \exists eo \in outtype(n, SF), \\ (tgt(eo) \in Y) \wedge step(n) \wedge fztime(T) \end{array} \right) \end{array} \right) \\ & \vee (Y = \emptyset \wedge run() \wedge fztime(T \setminus S) \wedge ((\exists n \in N : step(n)) \vee (\triangle(\emptyset) \wedge \Xi)) \end{aligned} \tag{5.23}$$

States, here s and s', correspond to tuples of the form $s = (mn, me, mnet, l_c, g_c, rec)$ and $s' = (mn', me', mnet', l'_c, g'_c, rec')$, whose elements are used in the definitions of St and Ct.

Definition 5.3.4 (Execution)**.** An execution is an infinite sequence of states such that $\sigma[0]$ is the initial state, and $\forall i \in \mathbb{N}, Next(\sigma[i], \sigma[i+1])$, where $\sigma[i]$ denotes the i^{th} state of the trace. Moreover, an execution is non-Zenon with regard to time and steps: there cannot be an infinite number of steps without time advancing, and there cannot be an infinite advancement of time without steps.

Formally, the non-Zenon hypothesis corresponds to weak-fairness on the left-hand part of the $Next$ disjunction and weak-fairness on its right-hand part. This ensures that if one node is enabled, it will eventually be done.

5.4 BPMN 2.0 and the Time Patterns: Can We Support All of Them?

The concept of a pattern was introduced by Christopher Alexander in [192] as *"The Timeless Way of Building"*. He defined a pattern as *"a three-part rule, which expresses a relation between a certain context, a problem, and a solution"*. Patterns characterise constructs, methods or techniques that have been encountered in practice repeatedly. Each pattern is intended to address an individual problem.

To solve more complex problems, a number of patterns may need to be combined. By classifying different patterns and the types of relations between them, patterns can more easily be combined. Moreover, with knowledge of the specific characteristics of individual patterns, one may choose the pattern most appropriate for a given situation. A pattern language helps a user move from problem to solution logically, thus allowing for many alternative paths through the design process. A pattern language is not fixed, it is built upon collected experience in a field, and as the techniques used in practice change, the pattern language may also evolve.

In the business process management field, some works have been done on identifying workflow patterns for different process perspectives control-flow, data and resources. Among them, we can find the *Workflow Patterns Initiative* [193] which collects more than 100 workflow patterns and compares various languages and tools based on their support for such patterns. The introduction of the workflow patterns has had a significant impact on PAIS design as well as on the evaluation of PAISs and process languages [194]. Our interest in pattern collections is relative to the time patterns. Several works propose approaches to deal with the time perspective of PAISs, such as a timed workflow process model in [195], process mining of temporal aspects in [196], verification of temporal constraints in [197–199], among others. However, the most complete and recent framework regarding time support in PAIS and considering time patterns in the literature was provided in [15]. The authors extend existing workflow patterns by describing time-related concepts commonly found in business processes and providing a reference system for them.

They present ten-time patterns representing temporal constraints commonly occurring in the context of time-aware processes. This section presents them all, and we discuss how BPMN supports seven of these patterns. For each pattern, we provide its description, its modelling using the BPMN standard w.r.t our interpretation, and we show how our formalisation deals with it.

5.4.1 Time Lags between Activities Pattern

Informal Description. It defines a minimal or maximal delay between two consecutive activities or both. The relation can be start-to-start (*i.e.*, between the start of two activities), start-to-end formalisation, between the start of the first and completion of the second activity), end-to-start (*i.e.*, between the completion of the first and the start of the second activity), or end-to-end (*i.e.*, between completion of two activities). [15]

Textual Specification. Using the BPMN standard, the time delay between activities may be presented using a $TICE$ or TBE and may support only the start-to-start and the end-to-start relations.

The relation end-to-start may be supported by a $TICE_p$ between two activities or by a $TBE_p^{\oslash}$ associated with an activity as presented in Figure 5.15. In contrast, the start-to-start relation requires two timers to be supported, where the first timer forces the start of the second timer. Therefore, we can model this pattern case using a $TICE_d$ or a TSE_d followed by an activity with $TBE_d^{\oslash}$ with the same time date as presented in Figure 5.15.

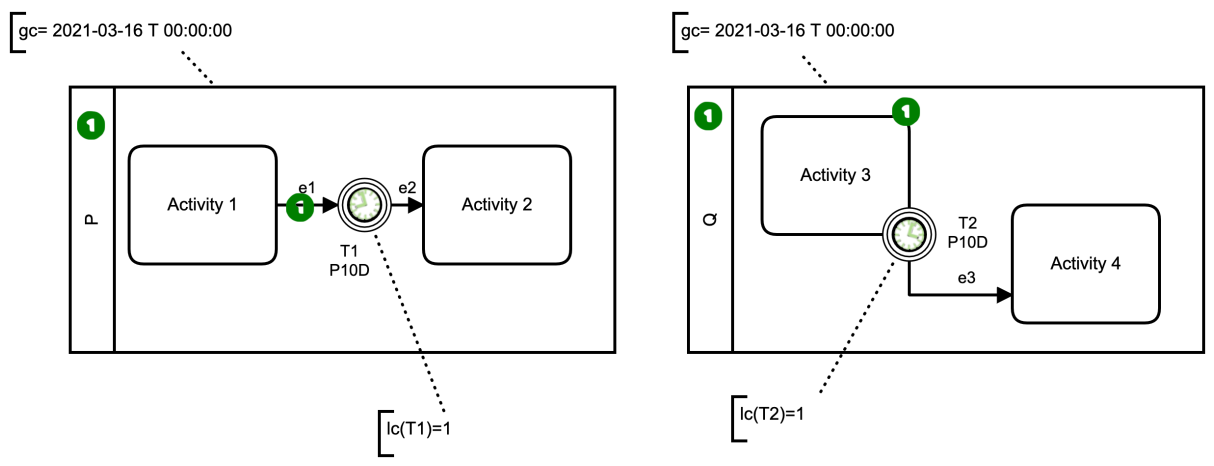


Figure 5.15: *Time Lags Pattern (End-to-Start).*

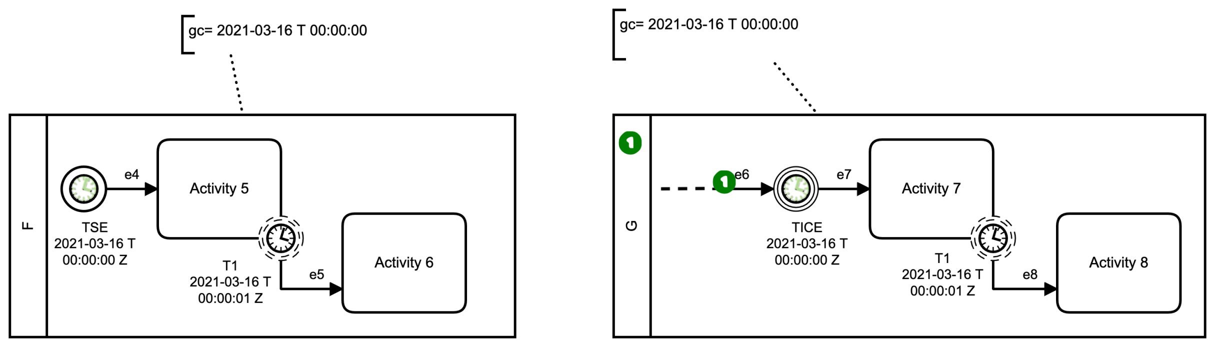


Figure 5.16: *Time Lags Pattern (Start-to-Start).*

Formal Semantics. Based on our textual representation, let us consider the states:

- $s_1 = < mn(P) = 1$, $me(e1) = 1$, $me(e2) = 0$, $mn(Activity\ 1) = 0$, $mn(Activity\ 2) = 0$, $l_c(T1) = 1$, $g_c =$ 2021-03-16 T 00:00:00, $rec = \iota >$ for the process P of Figure 5.15,

- s_2=< $mn(Q) = 1$, $me(e3) = 0$, $mn(Activity\ 3) = 1$, $mn(Activity\ 4) = 0$, $l_c(T2) = 1$, g_c = 2021-03-16 T 00:00:00, $rec = \iota$ > for the process Q of Figure 5.15,
- s_3=< $mn(F) = 0$, $me(e4) = 0$, $me(e5) = 0$, $mn(TSE) = 1$, $mn(Activity\ 5) = 0$, $mn(Activity\ 6) = 0$, $l_c(\emptyset) = 0$, g_c = 2021-03-16 T 00:00:00, $rec = \iota$ > for the process F of Figure 5.16,
- s_4=< $mn(G) = 1$, $me(e6) = 1$, $me(e7) = 0$, $me(e8) = 0$, $mn(TICE) = 0$, $mn(Activity\ 7) = 0$, $mn(Activity\ 8) = 0$, $l_c(\emptyset) = 0$, g_c = 2021-03-16 T 00:00:00,$rec = \iota$ > for the process G of Figure 5.16.

According to the form of the processes P, Q, F, G and the current states s_1, s_2, s_3 and s_4 of P's, Q's, F's, and G's instances, the execution evolves as follows:

- In Figure 5.15, the state of process P changes by executing the run() function (Formula 5.1) 864000 times (10 days in seconds). Then, the state of process P became s'_1=< $mn(P) = 1$, $me(e1) = 0$, $l_ce2 = 1$, $mn(Activity\ 1) = 0$, $mn(Activity\ 2) = 0$, $l_c(T1) = 0$, $g_c = 2021-03-26\ T\ 00:00:00$, $recrec(\emptyset) == \iota$ >. This execution state takes place by applying the completing predicate (Formula 5.5) of the $T1$ node, which requires the incoming edge $e1$ of the event to be marked by at least one token and the local clock of the event reaches its max $l_c(T1) = 864000$. Thus, the effects of the $T1$ execution are as follows: unmarked $e1$, marked $e2$, and reset the local clock $l_c(T1)$.
- With the same principle, the state of process Q of Figure 5.15 changes by executing the run() function (Formula 5.1) 864000 times (10 days in seconds). Then, the state of process Q became s'_2=< $mn(Q) = 1$, $me(e1) = 1$, $me(Activity\ 3) = 0$, $mn(Activity\ 4) = 0$, $g_c(T2) = 0$, $g_c = 2021-03-26\ T\ 00:00:00$, $rec(\emptyset) = \iota$ >. This execution state takes place by applying the completing predicate (Formula 5.7). This rule generates a token on edge $e3$. The latter enabled the execution of the abstract task starting predicate (Formula 5.15) for *Activity* 4, which results its activation.
- In Figure 5.16, the state of process F changes to s'_3=< $mn(F) = 1$, $me(e4) = 1$, $me(e5) = 0$, $mn(TSE) = 0$, $mn(Activity\ 5) = 0$, $mn(Activity\ 6) = 0$, $l_c(\emptyset) = 0$, $g_c = 2021-03-16\ T\ 00:00:00$, $rec(\emptyset) = \iota$ >. This execution state takes place by applying the completing predicate (Formula 5.3) of the timer start event (TSE). Next, this state enables the execution of the start predicate (Formula 5.15) of *Activity* 5. Then the state of the process F became s''_3=< $mn(F) = 1$, $me(e4) = 0$, $me(e5) = 0$, $mn(TSE) = 0$, $mn(Activity\ 5) = 1$, $mn(Activity\ 6) = 0$, $l_c(\emptyset) = 0$, g_c = 2021-03-16 T 00:00:01, $rec(\emptyset) = \iota$ >. This state allows for firing the timer event $T1$ and successively executing the predicates from formulas 5.8 and 5.15. Next, the F's state became s'''_3=< $mn(F) = 1$, $me(e4) = 0$, $me(e5) = 0$, $mn(TSE) = 0$, $mn(Activity\ 5) = 1$, $mn(Activity\ 6) = 1$, $l_c(\emptyset) = 0$, g_c = 2021-03-16 T 00:00:02, $rec(\emptyset) = \iota$ >.
- Same as process F, the state of the process G changes to s_4=< $mn(G) = 0$, $me(e6) = 0$,$me(e7) = 0$, $me(e8) = 0$, $mn(TICE) = 0$, $mn(Activity\ 7) = 1$, $mn(Activity\ 8) = 1$, $l_c(\emptyset) = 0$, g_c = 2021-03-16 T 00:00:02,$rec = \iota$ >. This execution state takes place by applying the predicates from Formulas 5.8, 5.15, 5.8, 5.15 successively.

5.4.2 Duration

Informal Description. It specifies a maximal duration for an activity or a whole process.

Textual Specification. Using the BPMN standard, the time duration for an activity may be represented by using an interrupting boundary event attached to an activity with a duration configuration (Figure 5.17, process P). However, to define a deadline for the whole process, we may use a parallel gateway between the entire process structure and the one that defines the duration deadline. This latter has an intermediate catch event with a duration configuration and ends with a terminate end event to enforce the cancelling of all the activities and events when the deadline reaches (Figure 5.17, process Q).

Formal Semantics. Based on our textual representation, let us consider the states:

- s_1=< $mn(P) = 1$, $me(e1) = 0$, $me(e2) = 0$, $mn(Activity1) = 1$, $mn(Activity2) = 0$, $mn(Activity3) = 0$, $l_c(T1) = 1$, g_c = 2021-03-17 T 00:00:00, $rec(\emptyset) = \iota$ > for the process P of Figure 5.17;
- s_2=< $mn(Q) = 1$, $me(e3) = 0$, $me(e4) = 1$, $me(e5) = 0$, $me(e6) = 1$, $me(e7) = 0$, $mn(Start) = 0$,

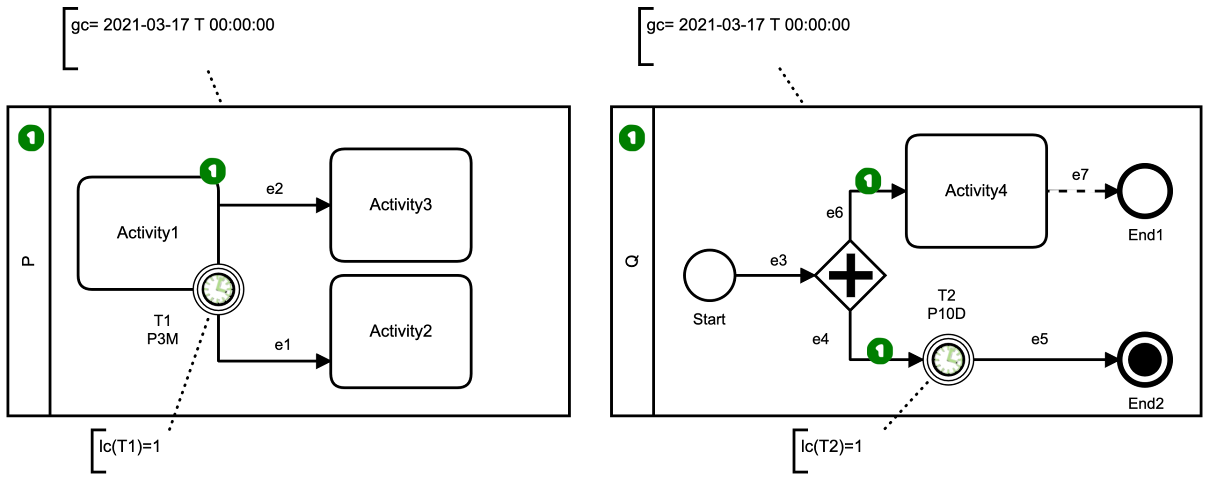


Figure 5.17: *Duration Time Pattern. (Maximum delay for activity (left) and Maximum delay for process (right))*

$mn(Activity4) = 0, mn(End1) = 0, mn(End2) = 0, l_c(T2) = 1, g_c =$ 2021-03-17 T 00:00:00, $rec(\emptyset) = \iota >$ and Q for the process Q of Figure 5.17.

According to the form of the processes P and Q and the current states s_1 and s_2 of P's and Q's instances, the execution evolves as follows:

- If the *activity*1 does not complete after 3 minutes ($P3M$), the state of process P changes by executing the run() function (Formula 5.1) 180 times. Then, the state of the process became $s_1' =< mn(P) = 1, me(e1) = 1, me(e2) = 0, mn(Activity1) = 0, mn(Activity2) = 0, mn(Activity3) = 0, l_c(T1) = 180, g_c =$ 2021-03-17 T 00:03:00, $rec(\emptyset) = \iota >$. This state takes place by applying the completing predicate of the completing predicate (Formula 5.7) of the $T1$ node, which requires that the activity it is attached to is active and the local clock reaches its max l_c(T1)=180. Thus, the effect of this execution stops *activity*1 at its time constraint.
- If the local clock of the process Q reaches its max time of 10 days ($l_c(T2) = 864000$), and there are active nodes in the process flow after *Activity*4, then, process Q state changes by executing the completing predicate (Formula 5.5) of the $T2$ node and terminating the whole process by executing the relevant rules.

5.4.3 Time Lags between Arbitrary Events Pattern

Informal Description. "It enables the specification of time lags between two discrete events. The latter may be related to the execution of activities but may also be triggered by an external source not controllable. [15]"

Textual Specification. Using the BPMN standard, the modelling of the discrete event that occurred based on specific conditions (*e.g.*, exception, messages, or errors, etc.) may be represented by intermediate throwing and catching nodes, boundary nodes attached to activities, or the activity itself defines a discrete event such as receiving a message.

Figure 5.18 shows an example for this pattern. It presents a case of two successive sending and receiving activities, where the maximum time lags between sending a request for a client service of an online shop and getting a response for this request should be no more than 48 hours.

Formal Semantics. Let us consider the state $s_1 =< me(P) = 1, me(Q) = 0, me(e1) = 0, me(e2) = 0, me(e3) = 0, me(e4) = 0, mn(Start) = 0, mn(End1) = 0, mn(End2) = 0, mn(Send) = 0, mn(Receive) = 1, l_c(T1) = 1, g_c =$ 2021-03-21 T 12:00:00, $rec(\emptyset) = \iota >$ where the process P has completed the sending of the request to the process Q and starts waiting for the response (the local clock $T1$ is activated). As long as the process P does not receive a message, the $run()$ function will be executed (Formula 5.1). If the local clock $T1$ reaches its maximum value, the state of process P changes to $s_1' =< mn(start) = 0, mn(end) = 0, me(e1) = 0, me(e2) = 0, me(e3) = 0, me(e4) = 0, me(e5) = 1,$

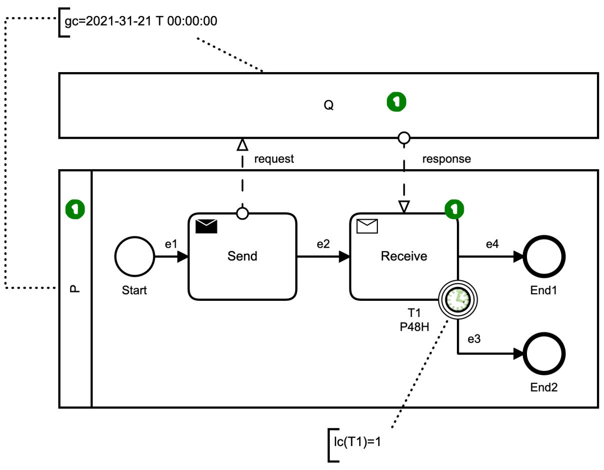


Figure 5.18: *Time Lags between Arbitrary Events.*

$mn(Send) = 0$, $mn(Receive) = 0$, $l_c(T1) = 0$, $g_c = 2021-03-23\ T\ 12:00:00$, $rec(\emptyset) = \iota >$ by executing the completing predicate of the boundary event $T1$ (Formula 5.6).

5.4.4 Fixed Date Elements (Deadline) Pattern

Informal Description. It allows specifying deadlines and fixed execution elements dates; it specifies a date deadline as an earliest (or latest) start (or completion) of an activity (or a process), *i.e.*, started after/before a fixed date, or complete before date. If the deadline is missed, the activity or process may never become active. [15]

Textual Specification. Using the BPMN standard, fixed date and time configuration may be used on different events (start or intermediate) and with different formats as presented above (see Section 5.2). For example, to represent the earliest fixed starting date for an activity, we may use an intermediate catch event with a timeDate configuration as a source node for its incoming edge (Figure 5.19, process P). On the other hand, to represent the latest completing fixed date for an activity, we may use an interrupting boundary event with a timeDate configuration (Figure 5.19, *Activity*2, process Q). Further, this latter may represent both the latest completion fixed date for an activity and the latest starting date for the successive activity (Figure 5.19, *Activity*3, process Q).

With the same principle, to represent a starting fixed date for a process, we may use a timer start event with a timeDate configuration (Figure 5.20, process P). Otherwise, to present its latest fixed date, we may use an intermediate catch event with a timeDate configuration in parallel with the process structure (Figure 5.20, process Q).

Formal Semantics. Let us consider the states:

- $s_1 =< me(P) = 1$, $me(e1) = 1$, $me(e2) = 0$, $me(e3) = 0$, $mn(Start) = 0$, $mn(Activity1) = 0$,

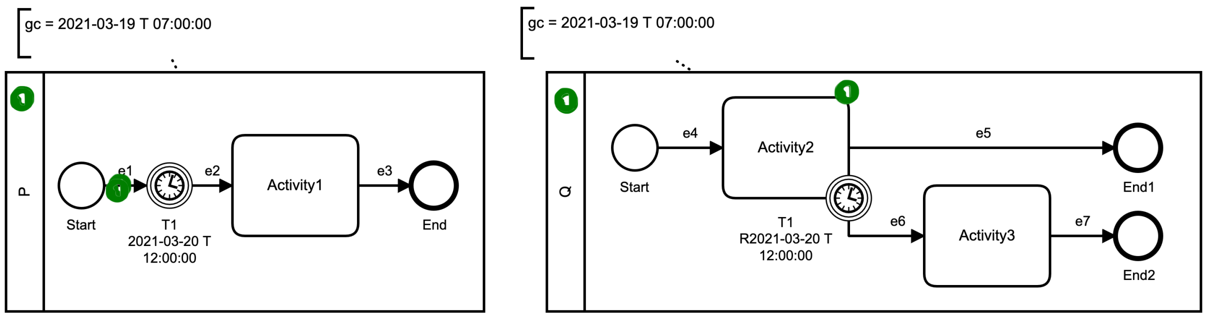


Figure 5.19: *Fixed Date Element Pattern for an Activity. Earliest start date for the Activity*1 *(P), Latest complete date for the Activity*2 *and Latest start date for the Activity*3 *(Q).*

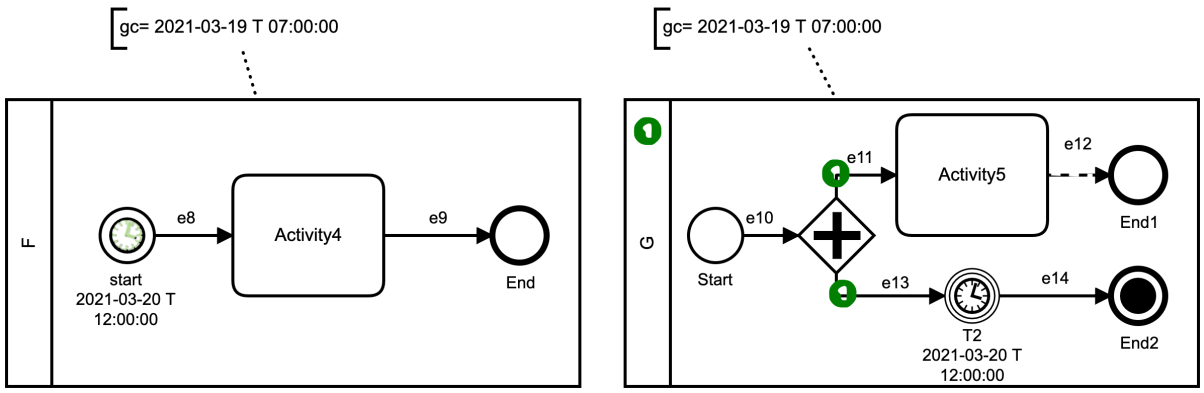


Figure 5.20: *Fixed Date Element Pattern for a Process. Latest start date (P) and Latest complete date (Q).*

$mn(End) = 0$, $l_c(\emptyset) = 0$, $g_c = 2021-03-19\ T\ 07:00:00$, $rec(\emptyset) = \iota >$ for the process P of Figure 5.19.

- $s_2 {=} < me(Q) = 1$, $me(e4) = 0$, $me(e5) = 0$, $me(e6) = 0$, $me(e7) = 0$, $mn(Start) = 0$, $mn(Activity2) = 1$, $mn(Activity3) = 1$, $mn(End1) = 0$, $mn(End2) = 0$, $l_c(\emptyset) = 0$, $g_c = 2021-03-19\ T\ 07:00:00, rec(\emptyset) = \iota >$ for the process Q of Figure 5.19.
- $s_3 {=} < me(F) = 0$, $me(e8) = 0$, $me(e9) = 0$, $mn(Start) = 1$, $mn(Activity4) = 0$, $mn(End) = 0$, $l_c(\emptyset) = 0$, $g_c = 2021-03-19\ T\ 07:00:00$, $rec(\emptyset) = \iota >$ for the process F of Figure 5.20.
- $s_4 {=} < me(G) = 1$, $me(e10) = 0$, $me(e11) = 1$, $me(e12) = 0$, $me(e13) = 1$, $me(e14) = 0$, $mn(Start) = 0$, $mn(Activity5) = 0$, $mn(End1) = 0$, $mn(End2) = 0$, $l_c(\emptyset) = 0$, $g_c = 2021-03-19\ T\ 07:00:00, rec(\emptyset) = \iota >$ for the process G of Figure 5.20.

According to the form of the processes P, Q, F, and G and the current states s_1 and s_2, s_3, s_4 of P's, Q's, F's, and G's instances, respectively, the execution evolves as follows:

- In Figure 5.19, the state of process P changes by executing the run() function (Formula 5.1) until the global clock reaches the timeDate $2021-03-20\ T\ 12:00:00$. The latter enables the execution of the timer node $T1$. Next, the state of the process changes by executing the completing predicate (Formula 5.4) of the $T1$ event, which requires that e_1 be marked and the global clock reaches its fixed date. The state of the process became $s'_1 {=} < me(P) = 1, me(e1) = 0, me(e2) = 1, me(e3) = 0$, $mn(start) = 0$, $mn(Activity1) = 0$, $mn(end) = 0$, $l_c(\emptyset) = 0$, $g_c = 2021-03-20\ T\ 12:00:00$, $rec(\emptyset) = \iota >$. This state enables, in its turn, the execution of $Activity1$.
- With the same principle, if $Activity2$ is still active and the global clock reaches the timeDate $2021-03-20\ T\ 12:00:00$, the state of process Q of Figure 5.19 changes by executing the completing predicate (Formula 5.6) of the $T1$ event, which requires that $Activity2$ be marked and the global clock reaches the fixed date. The state of the process became $s_2 {=} < me(Q) = 1, me(e4) = 0$, $me(e5) = 0$, $me(e6) = 1$, $me(e7) = 0$, $mn(Start) = 0$, $mn(Activity2) = 0$, $mn(Activity3) = 0$, $mn(End1) = 0$, $mn(End2) = 0$, $l_c(\emptyset) = 0$, $g_c = 2021-03-20\ T\ 12:00:00, rec(\emptyset) = \iota >$. This state enables, in its turn, the execution of $Activity3$ and so on.
- In Figure 5.19, when the global clock reaches the timeDate $2021-03-20\ T\ 12:00:00$, the state of process F changes to $s'_3 {=} < me(F) = 1$, $me(e8) = 1$, $me(e9) = 0$, $mn(Start) = 0$, $mn(Activity4) = 0$, $mn(End) = 0$, $l_c(\emptyset) = 0$, $g_c = 2021-03-20\ T\ 12:00:00$, $rec(\emptyset) = \iota >$ by executing the completing predicate (Formula 5.3) of the $Start$ node.
- With the same principle, If the global clock reaches the timeDate $2021-03-20\ T\ 12:00:00$ and there are active nodes in the process flow after the $Activity5$. Then, the state of process G changes to $s'_4 {=} < me(G) = 1, me(e10) = 0, me(e11) = 0, me(e12) = 0, me(e13) = 0, me(e14) = 1$, $mn(Start) = 0$, $mn(Activity5) = 0$, $mn(End1) = 0$, $mn(End2) = 0$, $l_c(\emptyset) = 0$, $g_c = 2021-03-20\ T\ 12:00:00, rec(\emptyset) = \iota >$ by executing the completing predicate of the $T1$ node (Formula 5.4). This state enables, in its turn, the execution of the $End2$ event.

5.4.5 Schedule Restricted Element Pattern

Informal Description. "It enables us to restrict the execution of a particular element by a schedule, i.e., a timetable (e.g., a bus schedule). The schedule itself is known at built-time, whereas the concrete dates are specified either at instantiation or run-time. The schedule provides restrictions on when the respective element can be executed. [15]"

Textual Specification. According to the authors, this pattern can be realised using a timer that is started at process instantiation time and expires at the first endpoint of one of the respective time slots. The timer is then reset, and its expiration date is set to the following endpoint of one of the time slots. This is repeated until the respective activity (process) has been started/completed or no more valid time slots are available according to the schedule. As an example of this pattern, between Paris and Algiers, there are flights at 6:05, 10:30, 12:25, 17:35 and 20:40. In this example, the period between (6:05 and 10:30) is different from the one between (10:30 and 12:25). Therefore, the presentation of such a pattern using the BPMN notation is not supported, as it needs a set of schedules with different periods.

5.4.6 Time Based Restrictions Pattern

Informal Description. It allows *"to restrict the number of times a particular process element can be executed within a predefined time. [15]"*. This restriction may be used for a number of concurrent executions or the number of executions per time period.

Textual Specification. BPMN offers the possibility to represent several executions between them a period using a cycle timeDefinition. Such a pattern may then be simulated using a timer catch event with a cycle time definition with a timeDuration including bounded/unbounded number of executions and conditional structure. *E.g.*, a subscriber may read only ten online articles per month for an annual subscription. If the number of readings is reached, no more books can be read in the current month (Figure 5.21).

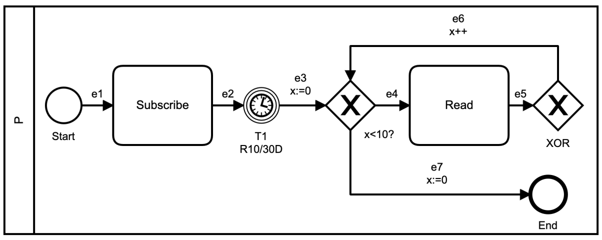


Figure 5.21: *Time Based Restrictions Pattern (Reading Article Example).*

Formal Semantics. As the time intermediate catch event with a cycle definition and the data treatment are out of the scope of this thesis. Therefore, we do not support the formalisation of this pattern.

5.4.7 Validity Period Pattern

Informal Description. It is similar to the pattern 5.4.4. It allows expressing that an activity or process must not be started or completed before or after a particular date. *I.e.*, its lifetime is restricted to this validity period. The respective process element may only be instantiated within a validity period.

Textual Specification. A validity period must be attached to the respective activity or process to realise this pattern respectively. Therefore, upon instantiation of the respective activity or process, its validity period needs to be checked. If the current time does not match with the activities (processes) validity period or the minimum duration of the activity (process) will result in the completion of the activity (process) being outside of the respective validity period, appropriate exception handling is required. This pattern is not supported using the BPMN notation.

5.4.8 Time Dependent Variability Pattern

Informal Description. "It allows varying control flow depending on the execution time or time lags between activities/events. [15]"

Textual Specification. The time-dependent variability may be represented in two different ways using the BPMN standard. The first one uses the boundary events (interrupting and non-interrupting) with varying formats of time to implement variability based on the activity time execution (Figure 5.22, right). The second one uses the deferred choice pattern [200], which is enabled by the use of the event-based gateway structure. This gateway allows the presentation of activities in combination with time triggers. The latter way implements the variability based on time lags between activities (Figure 5.22, left).
Formal Semantics. Based on our representation, the left-hand side of Figure 5.22 shows a collaboration diagram with two processes P and Q to represent a deferred choice pattern. The right-hand side of Figure 5.22 shows a collaboration diagram with three processes E, F and G to represent variability in the possible communication based on the timed execution of a process.

We represent the processes Q, F, and G with closed pools due to space restriction. Let us consider the states:

- $s_1 = < mn(P) = 1$, $me(e1) = 1$, $me(e2) = 0$, $me(e3) = 0$, $me(e4) = 0$, $me(e5) = 0$, $me(e6) = 0$, $me(e7) = 0$, $me(m1) = 0$, $me(m2) = 0$, $mn(StartP) = 0$, $mn(Activity1) = 0$, $mn(Activity2) = 0$, $mn(EndP1) = 0$,$mn(EndP1) = 2$, $mn(EndP3) = 0$, $l_c(T1) = 1$, $g_c = 2021-03-21\ T\ 12:00:00$, $rec(\emptyset) = \iota>$ or the process P of Figure 5.22;
- $s_2 = <mn(E) = 1$, $me(e8) = 0$, $me(e9) = 0$, $me(e10) = 0$, $me(e11) = 0$, $me(m1) = 0$, $me(m2) = 0$, $mn(StartE) = 0$, $mn(Activity3) = 1$, $mn(Activity4) = 0$, $mn(Activity5) = 0$, $mn(EndE) = 0$, $l_c(T2) = 1$, $g_c = 2021-03-21\ T\ 00:00:00$, $rec(\emptyset) = \iota >$ for the process E of Figure 5.22.

According to the form of the processes P and E and the current states s_1 and s_2 of P's and E's instances, respectively, the execution evolves as follows:

- As long as the process P does not messages, the run() function will be executed (Formula 5.1). If the local clock $T1$ node reaches its max of two days (l_c(T1)=172800), the state of the process P changes to $s_1' = < mn(P) = 1$, $me(e1) = 0$, $me(e2) = 0$, $me(e3) = 0$, $me(e4) = 1$, $me(e5) = 0$, $me(e6) = 0$, $me(e7) = 0$, $me(m1) = 0$, $me(m2) = 0$, $mn(StartP) = 0$, $mn(Activity1) = 0$, $mn(Activity2) = 0$, $mn(EndP1) = 0$,$mn(EndP1) = 2$, $mn(EndP3) = 0$, $l_c(T1) = 0$, $g_c = 2021-03-23\ T\ 12:00:00$, $rec(\emptyset) = \iota>$ by execution the completing predicate (Formula 5.14) of the EB node. The latter will be executed by choosing the alternative path of the timer node. Then the completing predicate (Formula 5.5) of $T1$ may be executed.
- The state of process E changes to $s_2 = <mn(E) = 1$, $me(e8) = 0$, $me(e9) = 0$, $me(e10) = 1$, $me(e11) = 0$, $me(m1) = 0$, $me(m2) = 0$, $mn(StartE) = 0$, $mn(Activity3) = 0$, $mn(Activity4) = 0$, $mn(Activity5) = 0$, $mn(EndE) = 0$, $l_c(T2) = 0$, $g_c = 2021-03-23\ T\ 00:00:00$, $rec(\emptyset) = \iota >$ by executing the starting predicate (Formula 5.7) of the $T2$ node. This state is reached if $Activity3$

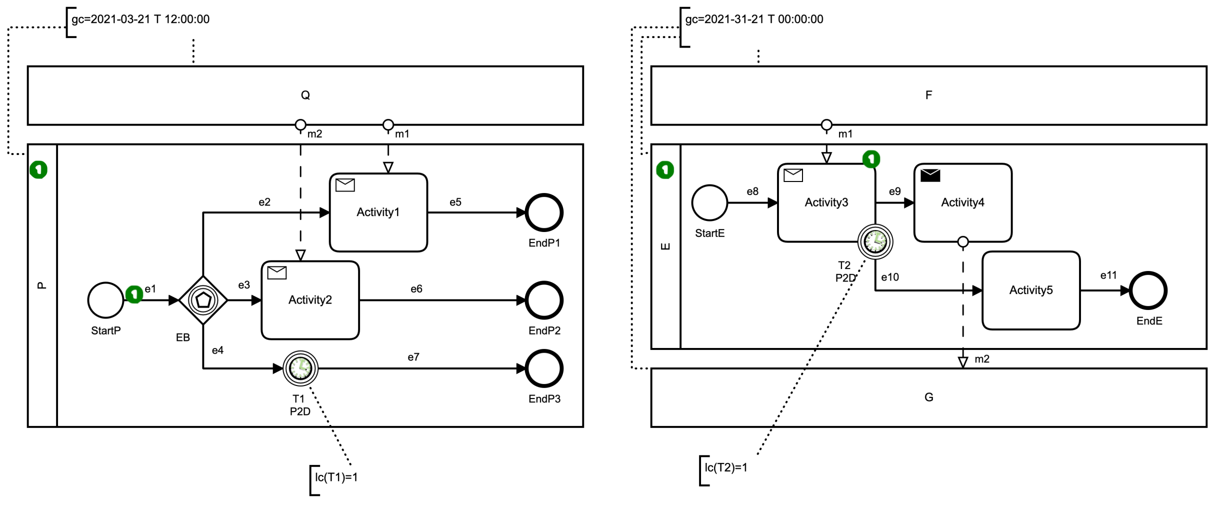


Figure 5.22: *Dependent Variability Pattern. Variability based on time lags between activities (left) and based on the time execution of an activity (right).*

has not been completed within two days, the alternative path of the timer node is then chosen, and no communication is done with process G.

5.4.9 Cycle Element Pattern

Informal Description. "A particular process element shall be iteratively performed with a time lags between the cycles. [15]". In such a pattern, the number of cycles is either fixed (*e.g.*, two cycles with an hour between them) or depends on some conditions. The time between the cycles may represent the minimum, maximum, or the time interval value and may realise start-to-start, start-to-end, end-to-start, and end-to-start relations between activities.

Textual Description. BPMN supports this pattern using a boundary timer event with cycle time category and duration configuration (Figure 5.23, process P), or a timer intermediate catch event with conditional gateways (Figure 5.23, process Q). The first one represents the relation start-to-start time lags between activities, and the second presents the end-to-start relation.

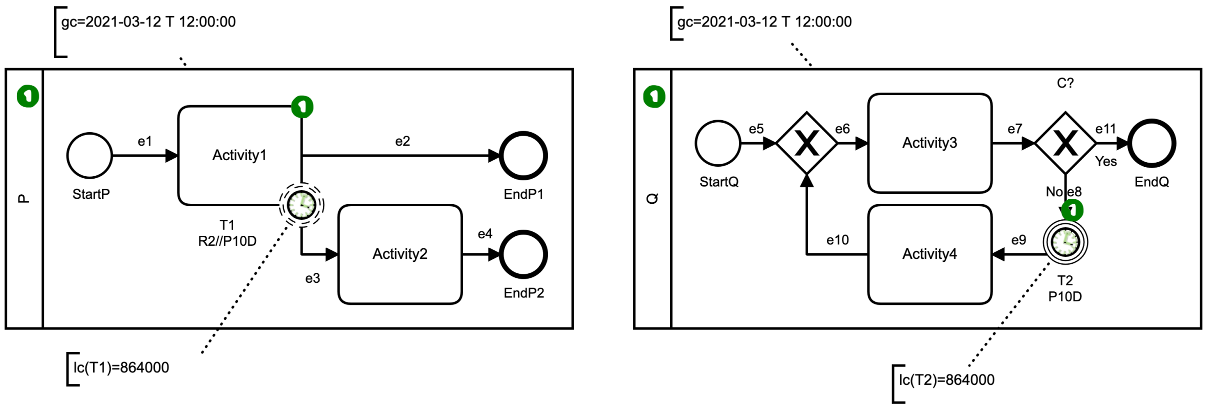


Figure 5.23: *Cycle Element Pattern. Cycle number being fixed (left) and cycle number depending on a condition C (right).*

Formal Semantics. Based on our formalisation, the data is abstracted. Thus, process Q execution is treated with non-determinism. Assuming the following states:

- $s_1 = <mn(P) = 1\ me(e1) = 0,\ me(e2) = 0,\ me(e3) = 0,\ me(e4) = 0,\ mn(StartP) = 0,\ mn(Activity1) = 1,\ mn(Activity2) = 0,\ mn(EndP1) = 0,\ mn(EndP2) = 0,\ l_c(T1) = 864000,\ g_c = 2021 - 03 - 12\ T\ 12 : 00 : 00,\ rec(T1) = 2>$ for the process P of Figure 5.23.
- $s_2 = <mn(Q) = 1\ me(e5) = 0,\ me(e6) = 0,\ me(e7) = 0,\ me(e8) = 1,\ me(e9) = 0,\ me(e10) = 0,\ me(e11) = 0,\ mn(StartQ) = 0,\ mn(Activity3) = 1,\ mn(Activity4) = 0,\ mn(EndQ) = 0,\ l_c(T2) = 864000,\ g_c = 2021 - 03 - 12\ T\ 12 : 00 : 00,\ rec(\emptyset) = \iota>$ for the process Q of Figure 5.23.

According to the form of the processes P and Q and the current states s_1 and s_2 of P's and Q's instances, respectively, the execution evolves as follows:

- The state of the process P changes to $s'_1 = <mn(P) = 1\ me(e1) = 0,\ me(e2) = 0,\ me(e3) = 1,\ me(e4) = 0,\ mn(StartP) = 0,\ mn(Activity1) = 1,\ mn(Activity2) = 0,\ mn(EndP1) = 0,\ mn(EndP2) = 0,\ l_c(T1) = 1,\ g_c = 2021 - 03 - 12\ T\ 12 : 00 : 00,\ rec(T1) = 1>$ by executing the staring predicate (Formula 5.13) of the non interrupting boundary event $T1$. The latter fires the timer node, resets the local clock of the timer node $T1$ to $l_c(T1) = 1$, and reduces the number of redundancy by 1. Then, as long as $Activity1$ doesn't complete, the run() function (Formula 5.1) is repeatedly executed, allowing the timer $T1$ to fire one more time. If the redundancy number is unbounded, $rec(T1) = \iota$, this behaviour is indefinitely repeated as long as the activity is active.
- The state of the process Q changes to $s'_2 = <mn(Q) = 1\ me(e5) = 0,\ me(e6) = 0,\ me(e7) = 0,\ me(e8) = 0,\ me(e9) = 1,\ me(e10) = 0,\ me(e11) = 0,\ mn(StartQ) = 0,\ mn(Activity3) = 1,\ mn(Activity4) = 0,\ mn(EndQ) = 0,\ l_c(T2) = 0,\ g_c = 2021 - 03 - 12\ T\ 12 : 00 : 00,\ rec(\emptyset) = \iota>$

by executing the staring predicate (Formula 5.5) of the intermediate catch event $T2$. This state enables, in its turn, the execution of $Activity4$ and $Activity3$ successively. Then, depending on the condition evaluation C (an undetermined choice in our case), a choice will be made to generate a token on $e1$ that activates the timer node $T2$ for the second time or not.

5.4.10 Periodicity Pattern

Informal Description. It allows specifying periodically recurring sets of activities according to an explicitly defined periodicity rule. This periodicity rule describes the recurrence schema of the respective activity (*e.g.*, group meetings will take place every two weeks at 11:30) as well as a particular start or exit condition (*e.g.*, starting from next Monday until the end of the year, five times). Unlike the cycle pattern (Section 5.4.9), this pattern emphasises possible execution dates of recurrent activities but not the time lag between them. Therefore, the periodicity rule may contain one or more than one date. [15]

Textual Description. BPMN standard supports this pattern by describing the time constraints with a timeCycle category. This category defines a repeated execution of the timer element with a fixed/dynamic number of iterations, depending on time lag and fixed start date, or time lag and fixed end date. Our work supports this configuration type associated with a non-interrupting event as illustrated in Figure 5.24, with two repetitions up to an ending date (process P) and two repetitions up after a starting date (process Q).

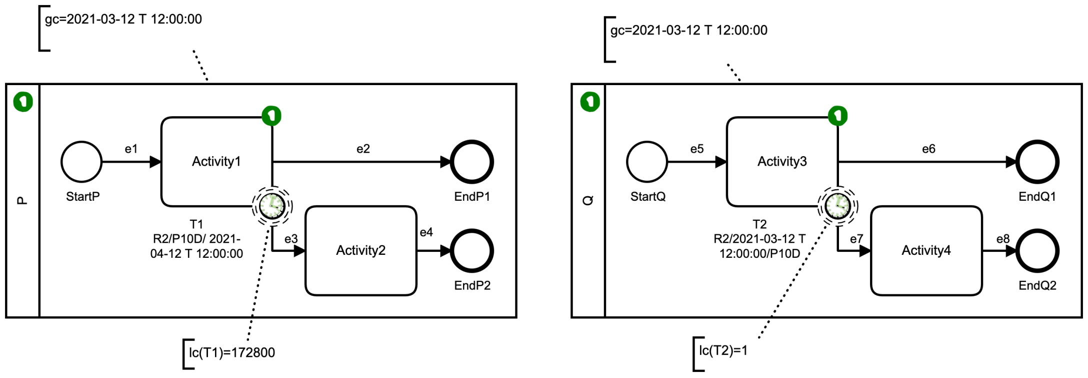


Figure 5.24: *Periodicity Pattern. Ending before a fixed date (left) and starting from a fixed date (right).*

Formal Semantics. Assuming the states:

- $s_1 = < mn(P) = 1, me(e1) = 0, me(e2) = 0, me(e3) = 0, me(e4) = 0, mn(Activity1) = 1, mn(Activity2) = 0, mn(EndP1) = 0, mn(EndP2) = 0, l_c(T1) = 172800, g_c = 2021-03-12\,T\,12:00:00, rec(T1) = 2>$ for the process P of Figure 5.24.
- $s_2 = < mn(Q) = 1, me(e5) = 0, me(e6) = 0, me(e7) = 0, me(e8) = 0, mn(Activity3) = 1, mn(Activity4) = 0, mn(EndQ1) = 0, mn(EndQ2) = 0, l_c(T2) = 1, g_c = 2021-03-12\,T\,12:00:00, rec(T2) = 2>$ for the process Q of Figure 5.24.

According to the form of the processes P and Q and the current states s_1 and s_2 of P's and Q's instances, respectively, the execution evolves as follows:

- The state of the process P changes to $s_1' = < mn(P) = 1, me(e1) = 0, me(e2) = 0, me(e3) = 1, me(e4) = 0, mn(Activity1) = 1, mn(Activity2) = 0, mn(EndP1) = 0, mn(EndP2) = 0, l_c(T1) = 1, g_c = 2021-03-12\,T\,12:00:00, rec(T1) = 1>$ by executing the staring predicate (Formula 5.12) of the non interrupting boundary event $T1$, The latter fires the timer node, resets the local clock of timer node $T1$ to $l_c(T1) = 1$, and reduces the number of redundancy by 1. Then, as long as $Activity1$ does not complete and the global clock does not reach the ending date, the run() function (Formula 5.1) is repeatedly executed, allowing the timer $T1$ to fire one more time.
- The state of the process Q changes to $s_2' = < mn(Q) = 1, me(e5) = 0, me(e6) = 0, me(e7) = 1, me(e8) = 0, mn(Activity3) = 1, mn(Activity4) = 0, mn(EndQ1) = 0, mn(EndQ2) = 0, l_c(T2) = 1, g_c = 2021-03-12\,T\,12:00:00, rec(T2) = 1>$ by executing the staring predicate

(Formula 5.10) of the non interrupting boundary event $T1$, which fires for the first time if the global clock reaches the starting date. It reactivates the local clock of timer node $T1$ to $l_c(T1) = 1$, and reduces the number of redundancy by 1. Then, as long as $Activity1$ does not complete, the run() function (Formula 5.1) is repeatedly executed, allowing the timer $T1$ to fire one more time by executing the completing predicate (Formula 5.11) of the non interrupting boundary event $T1$.

5.5 Summary

In this chapter, we have proposed formal semantics for the time-related constructs of BPMN with reference to the non-deterministic one in Chapter 4. This semantics supports different combinations of events, time information categories (date-times, durations, cycles) and the corresponding ISO-8601 descriptions as prescribed by the BPMN standard. Our proposal is based on a direct formalisation in First-Order Logic. In Table 5.2, we have presented the set of the supported timer elements and their relation to ISO-6801 standard. The table shows that we do not support: (i) the timeCycle type with a fixed interval of a start and end date due to the redundancy management ambiguity of this type; (ii) the timeCycle type for the starting event, which may lead to the execution of parallel multi-instance of a process (we do not support the multi-instance characteristics for the process node); (iii) the timeCycle type for the intermediate catching event. Note that the last one (iii) is under study as it is similar to the one presented for the non-interrupting boundary event.

Besides, we have studied the support with our formalisation of ten process time patterns defined in the literature for PAIS systems. First, we show that the BPMN standard may support the modelling of these patterns entirely or partially by considering the ISO-6801 standard time definitions. Then, we show that we support the formalisation of seven patterns out of the ten.

Part III

From Formal Semantics to Tool Support

Chapter

6 fbpmn: Formal BPMN Framework

“*Make it work.*
Make it work right.
Make it work right and fast.”
Edsger Dijkstra, Donald Knuth, C.A.R. Hoare

Chapter content

6.1 Introduction

The formal semantics introduced so far shows: how complex the modelling of a collaboration diagram can be, how the communication models vary and how to correctly figure out the execution of inter and intra-processes in the presence of messages, hierarchical structures, interrupting events, and time constructs. These make the overall behaviour of the collaboration challenging to grasp.

This chapter supports the presented formalisation with a tool-suite called fbpmn. This tool automatically verifies correctness properties for BPMN collaboration models and animates counterexamples when the properties are not satisfied. This tool is implemented to perform the verification using two formal specification languages, TLA$^+$ [17] and Alloy [46], that are increasingly popular due to their simplicity and flexibility, as well as the effectiveness of their companion model checkers, the TLC and Alloy Analyzer, respectively. These languages and tools are based on two different theories: TLA$^+$ is based on the explicit model checking, while Alloy is based on bounded model checking. All together, TLA$^+$ and Alloy languages support first-order logic properties.

This chapter explores these two frameworks for the verification of the business process properties. While TLC interprets TLA$^+$ models as finite state machines and deploys an explicit-state model checker, Alloy Analyser converts models into propositional formulas that are passed to SAT solvers. The properties of interest are encoded in the TLA$^+$ and Alloy theories (we have implemented). They include usual correctness properties for workflows as well as those (proposed more recently [140]) that are more specific to BPMN.

This chapter is organised as follows. Section 6.2 gives an overview of the fbpmn tool, its architecture and its general principles. Section 6.3 presents the encoding of the BPMN semantics in TLA$^+$. Section 6.4 presents the encoding of the BPMN semantics in Alloy language. Section 6.5 presents some results obtained with fbpmn tool on models, available on a publicly accessible repository. Section 6.6 provides the description of the tool's development and functionalities. Section 6.7 summarises the chapter.

6.2 fbpmn Overview

fbpmn (*Formal Business Process Modelling Notation*) is an open-source verification tool-chain software, based on our formalisation of BPMN collaboration diagrams, implemented using various TLA$^+$ and Alloy modules. For utilising the tool-chain, we provide a web interface at `http://vacs.enseeiht.fr/bpmn/` as well as a Virtual-Box virtual machine containing a local installation of our tool-chain at `https://github.com/pascalpoizat/fbpmn`. The overall overview of the fbpmn tool-suite is synthesised in Figure 6.1. Reading the figure from left to right, we have the following main components: Modelling Environnement, TLA Translator, Alloy Translator, TLA Modules, Alloy Modules, TLC Model checker, and Alloy Analyser. The figure shows an overview of the process for performing verification with the fbpmn tool. First, all the semantics rules we have defined in first-order logic are encoded in TLA (theories.tla files) and Alloy (theories.als files) languages (we will detail the content of these static models in the following sections). These files are written once for all. The Modelling Environment allows the design of BPMN models and selects properties to be verified. In principle, any BPMN editor can be used as such component, especially those compliant with the BPMN 2.0 standard such as Eclipse BPMN2 Modeler [201], Camunda Modeler [202] and Signavio Editor [203] (step (1), Figure 6.1). fbpmn takes as input a business process or a collaboration model in BPMN format. It starts by translating the model into a graph in terms of TLA$^+$ (resp. Alloy) encodings using the *TLA translator* (*Alloy translator*, respectively) (step (2), Figure 6.1). Choosing one of these translators generates a set of property and encoding model files (step (3), Figure 6.1). These generated files and theories files are then used as input for the checkers (TLC model checker or the Alloy analyser) to check the model (step (4), Figure 6.1). The latter returns results (step (6), Figure 6.1) that indicate the correctness or not of the properties on the model. fbpmn offers an animation option to generate counterexamples using these results (step (7), Figure 6.1). Indeed, it also provides the designer with the possibility to add properties manually (step (5), Figure 6.1).

In the following sections, we present : (i) our encoding of the FOL semantics in TLA$^+$ that allows one to easily parameter the communication properties and benefits from the efficient *TLC* model checker to verify collaborations automatically. (ii) Our encoding of the FOL semantics in Alloy allows verifying timed BPMN models.

6.3 Encoding of FOL Semantics in TLA

The expression and action fragments of TLA$^+$ are FOL-based, so the encoding of the semantics in TLA$^+$ is straightforward (459 lines of TLA$^+$ formulas).

The given FOL formalisation captures the behaviour of each of the BPMN components (nodes which can be events, activities, and gateways), and thus the behaviour of the whole BPMN model. This behaviour is defined using the concept of tokens which move from nodes to edges (and vice-versa) when specific conditions are fulfilled. The distribution of those tokens (marking) on the elements of the BPMN model describes its state. Hence, the whole behaviour is seen as a set of states reachable when specific transitions are fired.

As extensively described before, the idea here is to associate to each node (n) two predicates[1]: a first

[1]In some cases, the behaviour is described using only one of the two predicates.

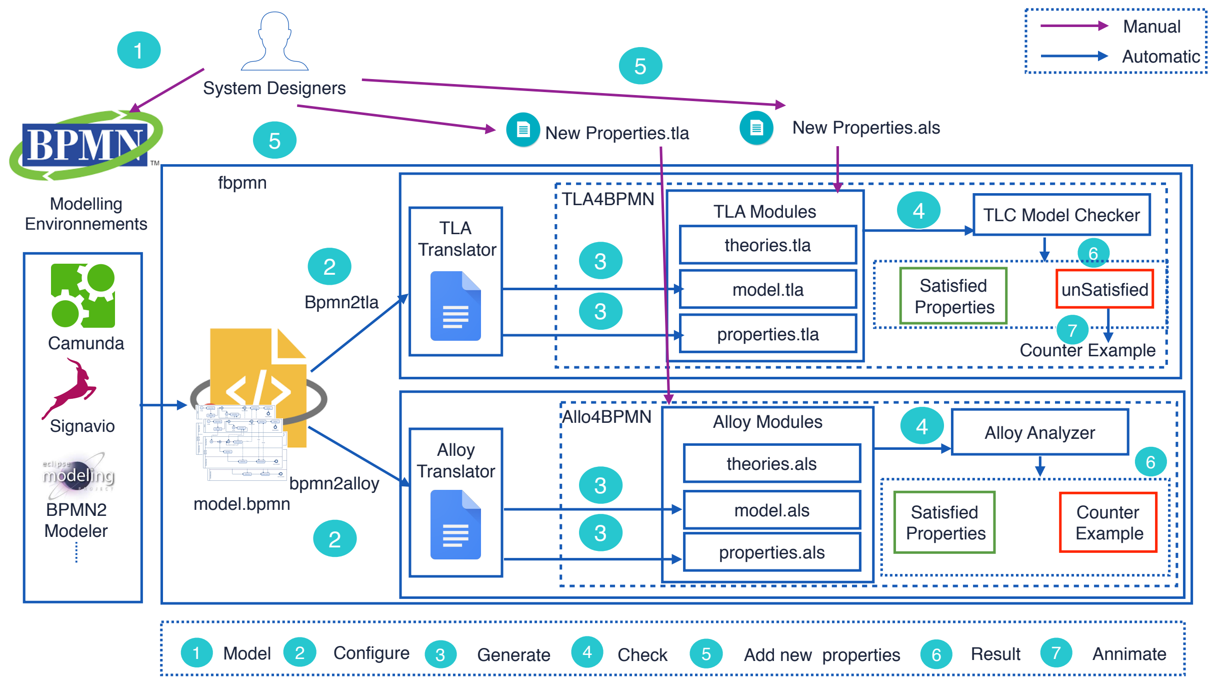


Figure 6.1: *Framework Overview.*

predicate, $St(\text{n})$, which states if the node can start its execution, and thus changing its current marking before its execution $(mn(n))$ to another marking after its execution $(mn'(n))$. The second predicate, $Ct(\text{n})$, that states if the node can finish its execution and so change the current marking $(mn(n))$ before its termination to another marking $(mn'(n))$ after its termination.

In this context, the TLA$^+$ specification of the semantics of node n is nothing but a direct translation of the $St(\text{n})$ and $Ct(\text{n})$ predicates of node n into the TLA$^+$ syntax.

Example.

Let us reconsider the semantics of a *None Start Event* defined through the predicate $Ct(\text{n})$:

$$
\begin{array}{rl}
\forall n \in N, Ct(n) \overset{def}{\equiv} & cat_N(n) = NSE \wedge (mn(n) >= 1) \wedge (mn'(n) = mn(n) - 1) \\
& \wedge\ \forall e \in outtype(n, SF), (me'(e) = me(e) + 1) \\
& \wedge \left(\begin{array}{l} \left(\begin{array}{l} \exists p \in N, p = R^{-1}(n) \\ \quad \wedge (cat_N(p) = P) \wedge (mn(p) = 0) \\ \quad\quad \wedge\ (mn'(p) = 1) \\ \quad \wedge \bigtriangleup (\{n, p\} \cup outtype(n, SF) \wedge \Xi) \end{array} \right) \\ \vee \left(\begin{array}{l} \exists sp \in N, sp = R^{-1}(n) \wedge (cat_N(sp) = SP) \\ \wedge \bigtriangleup (\{n\} \cup outtype(n, SF)) \wedge \Xi \end{array} \right) \end{array} \right)
\end{array}
$$

This FOL semantics is translated into a TLA$^+$ specification as follows.

$$
\begin{array}{rl}
nonestart\_complete(n) \triangleq & \wedge\ CatN[n] = NoneStartEvent \\
& \wedge\ nodemarks[n] >= 1 \\
& \wedge\ \text{LET } p == ContainRelInv(n) \text{ IN} \\
& \quad \vee\ \wedge\ CatN[p] = Process \\
& \quad\quad \wedge\ nodemarks[p] = 0 \\
& \quad\quad \wedge\ nodemarks' = [nodemarks \text{ EXCEPT } ![n] = @ - 1, ![p] = 1] \\
& \quad \vee\ \wedge\ CatN[p] = SubProcess \\
& \quad\quad \wedge\ nodemarks' = [nodemarks \text{ EXCEPT } ![n] = @ - 1] \\
& \wedge\ edgemarks' = [e \in \text{DOMAIN } edgemarks \mapsto \\
& \quad\quad \text{IF } e \in outtype(SeqFlowType, n) \text{ THEN } edgemarks[e] + 1 \\
& \quad\quad \text{ELSE } edgemarks[e]] \\
& \wedge\ Network!unchanged
\end{array}
$$

Table 6.1: *Translation between FOL and TLA$^+$ (NSE example).*

FOL Expression		TLA$^+$ expression	
ϕ_1	$cat_N(n) = NSE$	ϕ'_1	$CatN[n] = NoneStartEvent$
ϕ_2	$mn(n) \geq 1$	ϕ'_2	$nodemarks[n] >= 1$
ϕ_3	$\forall e \in outtype(n, SF),$ $(me'(e) = me(e) + 1)$	ϕ'_3	$edgemarks' = [e \in \text{DOMAIN } edgemarks \mapsto$ $\text{IF } e \in outtype(SeqFlowType, n)$ $\text{THEN } edgemarks[e] + 1$ $\text{ELSE } edgemarks[e]\ ]$
ϕ_4	$\exists p \in N, (p = R^{-1}(n))$ $\wedge\ (cat_N(p) = P)$ $\wedge\ (mn(p) = 0)$ $\wedge\ (mn'(n) = mn(n) - 1)$ $\wedge\ (mn'(p) = 1)$	ϕ'_4	LET $p == ContainRelInv(n)$ IN $\wedge\ CatN[p] = Process$ $\wedge\ nodemarks[p] = 0$ $\wedge\ nodemarks' =$ $[nodemarks \text{ EXCEPT } ![n] = @ - 1, ![p] = 1]$
ϕ_5	$\exists sp \in N, (sp = R^{-1}(n))$ $\wedge\ (cat_N(sp) = SP)$ $\wedge\ (mn'(n) = mn(n) - 1)$	ϕ'_5	LET $p == ContainRelInv(n)$ IN $\wedge\ CatN[p] = SubProcess$ $\wedge\ nodemarks' =$ $[nodemarks \text{ EXCEPT } ![n] = @ - 1]$
ϕ_6	Ξ	ϕ'_6	$Network!unchanged$

Intuitively, the translation is done syntactically, as shown in Table 6.1. Here, we can easily observe that the translation is straightforward, and this holds for all the elements of BPMN.

The resulting theories, for the translation of the whole elements, are available at `https://github.com/pascalpoizat/fbpmn` under `theories/tla`. They are:

Module PWSTypes, defines the abstract constants that correspond to the node and edge types.

Module PWSDefs, specifies the constants that describe a BPMN graph (Definition 4.2.1): `Node` (for N), `Edge` (for E), `Message` (for $\mathbb{M}$), `CatN` (for cat_N), `CatE` (for cat_E), `ContainRel` (for R), etc. This module also defines auxiliary functions such as $intype(type, n)$, defined in TLA$^+$ as the operator $intype(type, n) \triangleq \{e \in \{ee \in Edge : target(ee) = n\} : CatE[e] \in type\}$.

Module PWSWellFormed, encodes the well-formedness predicates for BPMN graphs. For instance, the rule C3 (a sub-process has a unique start event node) becomes:

$$C3\_SubProcessUniqueStart \triangleq \forall n \in Node : CatN[n] = SubProcess \Rightarrow$$
$$Cardinality(ContainRel[n]\ \cap \{nn \in Node : CatN[nn] \in StartEventType\}) = 1$$

Module PWSSemantics, contains the semantics. It defines the variables for the marking: `nodemarks` ($\in [Node \to Nat]$), `edgemarks` ($\in [Edge \to Nat]$), and `net` (whose type depends on the selected communication model). Then it contains a translation of the FOL formulas, where each rule yields one TLA$^+$ action, translated from the FOL semantics as explained above. The $Next$ predicate specifies a possible transition between a starting state and a successor state. It is a disjunction of all the actions. The full specification is then, as usual in TLA$^+$, $Init \wedge \Box[Next]_{var} \wedge Fairness$, where $Init$ specifies the initial state (Definition. 6.4), and $\Box[Next]$ specifies that $Next$ (or stuttering) is verified along all the execution steps. Further, the restriction to fair executions (Section 4.4.0.5) is naturally translated in TLA$^+$. TLA$^+$ supports weak and strong fairness, defined as below for an action A:

$$WF_e(\mathcal{A}) \overset{def}{\equiv} \Box\Diamond\neg(\text{ENABLED}\langle\mathcal{A}\rangle_e) \vee \Box\Diamond\langle\mathcal{A}\rangle_e$$
$$SF_e(\mathcal{A}) \overset{def}{\equiv} \Diamond\Box\neg(\text{ENABLED}\langle\mathcal{A}\rangle_e) \vee \Box\Diamond\langle\mathcal{A}\rangle_e$$

Fairness is then a conjunction of weak fairness on all actions ($\forall n \in Node : WF_{var}(step(n))$), and of strong fairness on XOR, OR and EB transitions.

6.3.1 Communication as a Parameter.

One of the objectives of our FOL semantics is to be able to specify the communication behaviour as a parameter of the verification. To achieve this, all operations related to communication are isolated in a `Network` module. This module is a proxy for several implementations corresponding to communication models with different properties, such as their delivery order.

Generic Communication Models We provide seven communication models which differ in the order messages can be sent or received and are all the possible point-to-point models when considering local ordering (per process), causal ordering, and global ordering (absolute time). Their formal description is provided in section 4.3.3, and their formal analysis and comparison can be found in [179].

The state of the communication model is specified with a variable `net`, whose content depends on the communication model. The communication actions are two transition predicates `send` and `receive` which are true when the action is enabled. These actions take three parameters, the sender process, the destination process and the type of message. Their specification depends on the communication model and is a direct translation in TLA$^+$ of the FOL formula of table 4.3. For instance, `NetworkFifo` specifies a communication model where the delivery order is globally first-in-first-out: messages are delivered in the order they have been sent. Its realisation is a queue, and the two predicates are:

$$\begin{aligned} send(from, to, m) &\triangleq net' = Append(net, \langle from, to, m\rangle) \\ receive(from, to, m) &\triangleq net \neq \langle\rangle \wedge \langle from, to, m\rangle = Head(net) \wedge net' = Tail(net) \end{aligned}$$

Ad-hoc Communication Models The communication models described in 4.3.3 are all monolithic. This means that all the communication interactions are handled by the same communication model and that it restricts the receptions in the same way for all communication channels. In some cases, one needs to have different properties in different parts of a model. For instance, a set of processes can require FIFO All communication for their interactions, while another set does not have any constraint. Using a modular communication framework based on micromodels [204], we offer the possibility to implement the *send* and *receive* predicates with a combination of micro models that are applied to subsets of the channels in the BPMN collaboration. The available micromodels are the seven ordering model as above, that order the receptions with regard to the emission events; a micromodel where priorities are assigned to channels; a message capacity micromodel that limits the number of messages in transit; a bounded micromodel that limits the total number of messages that a set of channels can transport in an execution.

Consider the example in Figure. 2.8. This example has an infinite number of states as the travel agency can send an arbitrary number of the offer. Moreover, it is required that the `NoMore` message is received after all the `Offer` messages. With the monolithic communication models, this can be handled by using the FIFO Pair (or FIFO All) communication model. However, observe that the `Confirmation` and `Ticket` messages are expected by the customer in the reverse order of their emissions. Imposing FIFO ordering means that the ticket cannot be delivered before the confirmation, and this collaboration with a FIFO communication model is unsound: the customer process deadlocks without reaching the final state.

```
CHANNELS == {"Offer", "NoMore", "Travel", "Payment", "Abort",  , "Ticket", "Confirmation"}
COMMODELS == {[name ↦ "p2p", params ↦ [chan ↦ CHANNELS ] ],
                    [name ↦ "fifo11", params ↦ [chan ↦ {"Offer","NoMore"} ] ],
                    [name ↦ "voting", params ↦ [chan ↦ {"Offer"}, bound ↦ 2 ] ],
                    [name ↦ "message_cap", params ↦ [chan ↦ CHANNELS, bound ↦ 4 ] ] }
COM == INSTANCE multicom WITH
                    PEERS ← {"Customer", "Travel Agency"}
                    COM ← COMMODELS,
                    CHANNEL ← CHANNELS
```

6.3.2 Mechanised Verification

A specific BPMN diagram is described by instantiating the constants in `PWSDefs` (`Node`, `Edge`...) from the BPMN collaboration. This is automated using our `fbpmn` tool. Regarding the well-formedness of the BPMN diagram, the predicates from `PWSWellFormed` are *assumed* in the model. Before checking a model, TLA$^+$ model checker checks these assumptions with the instantiated constants that describe the diagram and reports an error if an assumption is violated.

TLA$^+$ model checker, TLC, is an explicit-state model checker that checks both safety and liveness properties specified in LTL. This logic includes operators $\Box$ and $\Diamond$ that respectively denote that, in all executions, a property P must always hold ($\Box P$) or that it must hold at some instant in the future ($\Diamond P$). TLC builds and explores the full state space of the diagram to verify if the given properties are verified. These properties are generic properties (related to any business process diagram) or specific properties (for a given diagram). Examples of generic properties are: safeness, soundness and message-relaxed soundness [23].

A collaboration is safe if no sequence flow holds more than one token:

$$\Box(\forall e \in E, (cat_E(e) = SF) \land (me(e) \leq 1)) \tag{6.1}$$

A collaboration is sound if all the processes are sound and there are no undelivered messages. A process is sound if it is in a stable state where there are no tokens on its inside edges, and no tokens on its nodes, except possibly for start and end events.

$$\begin{aligned} SoundProc(p) &\overset{def}{=} \forall e \in R(p) \cap E, (cat_E(e) = SF) \land (me(e) = 0) \\ &\qquad \land \forall n \in R(p) \cap N, (mn(n) = 0 \lor (mn(n) = 1 \land cat_N(n) \in (EE \cup SE)) \\ Soundness &\overset{def}{=} \Diamond\Box(\forall p \in N, cat_N(p) = P \land SoundProc(p) \land \forall e \in E, cat_E(e) = MF \land me(e) = 0) \end{aligned} \tag{6.2}$$

A collaboration is a message-relaxed sound if it is sound when ignoring messages in transit, *i.e.*, when ignoring the Message Flow edges.

$$MsgRelaxedSoundness \overset{def}{=} \Diamond\Box(\forall p \in N, cat_N(p) = P \land SoundProc(p)) \tag{6.3}$$

Other generic properties are available, such as the absence of undelivered messages or the possible activation, which states that there does not exist a task node (Abstract Task, Send Task, Receive Task) that is never activated in any execution. From a business process point of view, it means that there are no tasks in the diagram that are never used. This is expressed by $\forall n \in N, cat_N(n) = T : \Diamond(mn(n) = 0)$. Actually, the invalidity of this formula confirms the satisfiability of its negation.

Last, the user can also define business model properties concerning a specific diagram. For instance, one can check that the marking of a given node is bounded by a constant *e.g.*,

$$(\Box(nodemarks[\text{"}\mathit{Confirm\ Booking}\text{"}]) \leq 1)$$

or that the activation of one node necessarily leads to the activation of another node *e.g.*,

$$(\Box(nodemarks[\text{"}\mathit{Book\ Travel}\text{"}] \neq 0 \Rightarrow \Diamond(nodemarks[\text{"}\mathit{Offer\ Completed}\text{"}] \neq 0))$$

Termination of the verification is ensured for a finite state model. When the model checker finds that a property is invalid, it outputs a counter-example trace that we animate on the BPM graphical model to help the user understand it. TLC uses a breadth-first algorithm, and this trace is minimal for safety properties. As any BPMN model is structurally finite, a model with an infinite state space is necessarily unsafe (in the sense of (6.1): some edges hold more than one token). This property is invalidated on a prefix of a trace. TLC incrementally checks invariants during the state space construction, and an unsafe model will be detected even if it would yield an infinite state space. TLC cannot check arbitrary properties on an infinite state model. Nevertheless, we can use constraints expressed on states or transitions to limit the state space (see Section 6.6.4.1).

6.4 Encoding of the Semantics in Alloy

This section details the different Alloy modules and routines to check the business process model automatically. Generally, Alloy does not provide a pre-defined way to model dynamic behaviour. Therefore, it is necessary to use the idiom *traces pattern* [47]. This introduces a signature to represent the system's overall state and model operations as predicates. Thus defining the relationships between the states before and after. We use the *Global State* variant of this pattern.

The signature and the predicate fragments of Alloy is also based on FOL. Hence, the encoding of the semantics in Alloy is straightforward (670 lines of Alloy formulas). An *abstract signature* defines an element type (node or edge), and the subtype relation relates these signatures (*e.g.*, $Node \supseteq Event \supseteq IntermediateEvent \supseteq TICE \supseteq TICE_d$). A BPMN graph combines these signatures with unique elements that correspond to the graph nodes, edges, and attributes that mark the endpoints of an edge.

As extensively discusssed for the TLA$^+$ translation, each semantic rule presented in Section 5.3 yields a *predicate* syntactically identical to it.

Example.
Let us reconsider the semantics of a *Timer Start* event that may be specified with a date-time (see equation. 6.4).

$$
\begin{aligned}
Ct(n) \overset{def}{\equiv} \; & (cat_N(n) = TSE_d) \wedge (ftime(n) \downharpoonright_{timeVal_D} = g_c) \\
& \wedge (mn(n) = 1) \wedge (mn'(n) = mn(n) - 1) \\
& \wedge \forall\ eo \in outtype(n, SF), (me'(eo) = me(eo) + 1) \\
& \wedge \exists\ p \in N, cat_N(n) = P, n \in R(p), (mn(p) = 0) \wedge (mn'(p) = mn(p) + 1) \\
& \wedge\ \triangle\,(\{n, p\} \cup outtype(n, SF)) \wedge \triangle_t(\emptyset)
\end{aligned}
\tag{6.4}
$$

This predicate is translated into Alloy as follows (see Listing 6.1).

```
 // ...

 pred State.canfire[n : TimerStartEvent] {
  n.mode in Date ∧this.globalclock ≥n.mode.date
 }
pred completeTimerStartEvent[s, s' : State, n: TimerStartEvent] {
  s.nodemarks[n] >0
  s.canfire[n]
  s'.nodemarks[n] =s.nodemarks[n].dec
  all~e : n.outtype[SequentialFlow] |s'.edgemarks[e] =s.edgemarks[e].inc
  let~p =n.~contains {
    s'.nodemarks[p] =s.nodemarks[p].inc
    delta[s, s', n + p, n.outtype[SequentialFlow]]
    deltaT[s, s', none] // localclock is unused
  }
}
```

Listing 6.1: *An Excerpt of the Timer Start Event Semantics Implementation in the Alloy Language.*

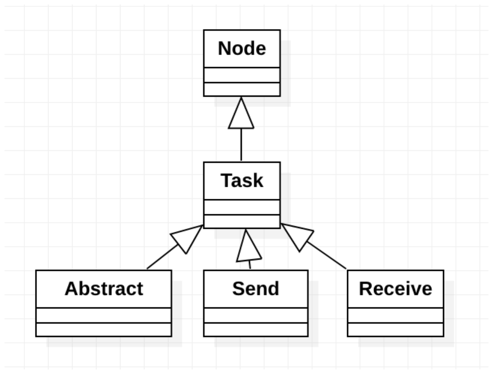
The translation is syntactical, as shown in Table 6.2. Here, we can easily observe that the translation is straightforward, and this holds for all the elements of BPMN.

Table 6.2: *Translation between FOL and Alloy (TSE example).*

	FOL Expression		TLA$^+$ expression
ϕ_1	$cat_N(n) = TSE_d$	ϕ'_1	$n : TimerStartEvent$
ϕ_2	$ftime(n) \downharpoonright_{timeVal_D} = g_c$	ϕ'_2	$n.mode\ in\ Date$ $\&\&\ this.globalclock\ >=\ n.mode.date$
ϕ_3	$mn(n) > 0$	ϕ'_3	$s.nodemarks[n] > 0$
ϕ_4	$\forall\ eo \in outtype(n, SF)$ $(me'(eo) = me(eo) + 1)$	ϕ'_4	$all\ e : n.outtype[SequentialFlow]$ $\vert s'.edgemarks[e] = s.edgemarks[e].inc$
ϕ_5	$\exists p \in N, cat_N(n) = P, (n \in R(p))$ $\wedge\ (mn(p) = 0)$ $\wedge\ (mn'(n) = mn(n) - 1)$ $\wedge\ (mn'(p) = mn(p) + 1)$	ϕ'_5	$let\ p = n.\ contains$ $s.nodemarks[p] = 0$ $s'.nodemarks[n] = s.nodemarks[n].dec$ $s'.nodemarks[p] = s.nodemarks[p].inc$
ϕ_6	$\triangle(\{n, p\} \cup outtype(n, SF))$ $\triangle_t(\emptyset)$	ϕ'_6	$delta[s, s', n + p, n.outtype[SequentialFlow]]$ $deltaT[s, s', none]//localclockisunused$

The resulting theories are available in the fbpmn repository at `https://github.com/pascalpoizat/fbpmn/tree/master/theories/alloy` under `theories/alloy`. The latter has a set of *static* and *dynamic* modules. `Static modules` are those writing once for all. They are:

Module PWSSyntax.als, It represents the BPMN elements syntax (edges, nodes, and time definition types). More specifically, this module contains a set of Alloy signatures for the BPMN elements needed to describe a process instance. For example, the listing in Figure 6.2 shows a set of abstract signatures focusing on the BPMN meta-model's *Activity* node types. Thanks to Alloy's object-oriented notation, it is natural and direct to represent a meta-model. Therefore, each metaclass corresponds to a signature and the attributes of metaclasses to relationships.



```
/** Activities **/
abstract sig Task extends Node {}
abstract sig AbstractTask extends Task {}
abstract sig SendTask extends Task {}
abstract sig ReceiveTask extends Task {}
```

Figure 6.2: *An Expert of the Alloy Implementation from the Syntax.als Module Represent the Activity Signature.*

Further, *Timer* nodes have a constraint that can be a date, a duration, or a duration with a repetition factor. In this last case, it can also have a starting date xor an ending date (TSE can only have a *Date*, $TICE$ can have a *Date* or *Duration*, and TBE any one of them). These time types are translated in Alloy by a set of abstract signatures with a `mode` ($\in$ *Ctime*) attribute. Listing 6.2 shows an excerpt of the implementation for the timer structure.

```
abstract sig Date { date : one Int }
abstract sig Duration { duration : one Int }
abstract sig CycleDuration extends Duration { repetition : one Int }
abstract sig CycleStart extends CycleDuration { startdate : one Int }
abstract sig CycleEnd extends CycleDuration { enddate : one Int }
abstract sig TimerStartEvent extends StartEvent { mode : one Date}
// ...
```

Listing 6.2: *An Excerpt of the Implementation of Time Structure.*

Module PWSDefs.als, specifies the constants that describe the BPMN graph (Definition 5.2.1) and a set of auxiliary functions.

Module PWSWellformed.als, encodes the well-formedness predicates for the BPMN presented in (Section 4.2.3).

Module PWSSemantics.als, contains the translation of the semantics rules of each element presented in Chapter 4.4 and those of Chapter 5.3. Each rule yields an Alloy predicate. As idiomatic in Alloy, execution is an ordered set of *States*, where a *fact* (a constraint that always holds) relates two consecutive states (in this ordering). Listing 6.3 presents the state signature used to represent a system state. Line 1, the *UTIL/ORDERING* module is used to create a total linear order between the different states. Each relation of the State signature (nodemarks, edgemarks, network, globalclock, and localclock) corresponds to a piece of state information as defined in the formalisation (see State Definition 5.3.1).

```
1  open util/ordering[State]
2    sig State {
3      nodemarks : Node -> one Int,
4      edgemarks : Edge -> one Int,
5      network : set (Message -> Process -> Process),
6      globalclock : one Int,
7      localclock : (TimerStartEvent + TimerIntermediateEvent + TimerBoundaryEvent) -> one Int,
8    } // ...
```

Listing 6.3: *State Implementation in Alloy.*

Listing 6.4 shows the initial state predicate defined in the formalisation (see Initial State Definition 5.3.2). This predicate gives the first State of the system. Line 10 shows the fact calling the *initialState* predicate, allowing to initialise the first state of the system.

```
1   pred initialState [globalC : Int]{
2     first.edgemarks =(Edge -> 0)
3     let processNSE ={ n : NoneStartEvent + TimerStartEvent |n.containInv in Process } {
4       first.nodemarks =(Node -> 0) ++ (processNSE -> 1)
5     }
6     first.network =networkinit
7     first.globalclock =globalC
8     first.localclock =(TimerStartEvent + TimerIntermediateEvent + TimerBoundaryEvent) -> 0
9   }
10  fact init { initialState }
```

Listing 6.4: *Initial State Predicate Implementation in Alloy.*

To simplify the trace predicate, it is possible to define different functions and predicates, as shown in Listing 6.5. The predicate *step* defines the set of execution steps for all the node types. It takes as parameters s corresponding to the current state and s' corresponding to its successor state. The predicate *deadlock* defines the state where no node may be executed. The predicate *someTimerIsActive* determines whether a given timer node is waiting to move the time forward and not ready to fire. Lastly, the predicate *advancetime* determines when the time moves forward the global and the local clocks.

```
pred step[s, s' : State, n: Node] {
  n in AbstractTask implies { startAbstractTask[s,s',n] or completeAbstractTask[s,s',n] }
  else n in SendTask implies { startSendTask[s,s',n] or completeSendTask[s,s',n] }
  completeMessageStartEvent[s,s',n] }
  //.....
}
// ...................
pred State.someTimerIsActive {
  // easy case: a local clock is counting
  { some n : TimerStartEvent |this.localclock[n] >0 ∧not this.canfire[n] }
  or { some n : TimerIntermediateEvent |this.localclock[n] >0 ∧not this.canfire[n] }
  //...
}
pred State.deadlock {
  no n : Node {
    this.canstartAbstractTask[n]
    or this.cancompleteAbstractTask[n]
// ...................
  }
}
// ...................
pred advancetime[s, s': State] {
all n : TimerStartEvent + TimerIntermediateEvent + TimerBoundaryEvent {
  s.localclock[n] >0 implies s'.localclock[n] =s.localclock[n].inc
  else s'.localclock[n] =s.localclock[n]
}
s'.globalclock =s.globalclock.inc
           }
```

Listing 6.5: *An Excerpt of Predicates Implementation in Alloy.*

Finally, as we are running bounded model-checking, we must ensure that enough steps are realized. Formally, with infinite executions, weak fairness is sufficient. As we integrate the time notion with the bounded model-checking, if moving the time forward is always possible, the execution may always take it and waste a number of steps. Our solution is to move time only if no node may be executed (deadlock) or there is a timer node still active and not ready to complete. Listing 6.6 shows the fact traces that constrains all the states. This fact is our predicate *Next* (Definition 5.3.3). It represents a disjunction of the semantic rules and of time moving.

```
    fact traces {
    all s: State - last {
      { (s.deadlock or s.someTimerIsActive) ∧~ delta[s, s.next, none, none] ∧~ advancetime[s, s.next] }
      or
      { some n : Node - Process |step[s, s.next, n] }
    }
  }
```

Listing 6.6: *An Excerpt of Next Predicate Implementation in Alloy.*

Module PWSProp.als, contains a set of properties defined in Section 6.4.1.

Conversely, the `dynamic modules` are those generated according to the business process model and the properties to analyse.

ProcessModel.als, encodes the process instance to be analysed. Figure 6.3 shows a simple process with interrupting boundary time date event. *Alloy Translator* component (cf. Figure 6.1) allows generating an Alloy specification as shown in Listing 6.7 for the given simple process model file. Each process element is represented as a singleton (one sig) of its equivalent type, defined in *PWSSyntax.als* module. As highlighted in this listing, the representation of the process in Alloy language is natural and straightforward. Each element and concept corresponds to a new instantiation of a signature defined in the *PWSSyntax.als* module.

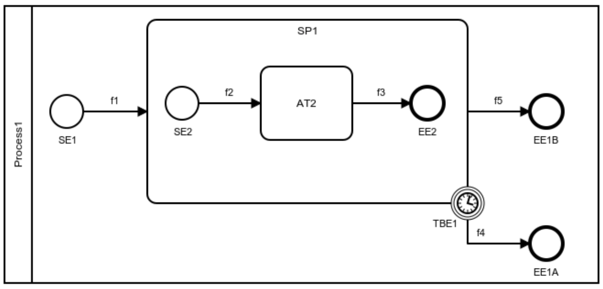


Figure 6.3: *A Simple Process Example with Time Date Constraint.*

```
module example_TBEI_SP
open PWSSyntax
open PWSSemantics
one sig SE1 extends NoneStartEvent {}
one sig SP1 extends SubProcess {} { contains =SE3 + AT3 + EE3}
one sig TBE1time extends Date {} {date =4}
one sig TBE1 extends TimerBoundaryEvent {} {
  attachedTo =SP1
  interrupting =True
  mode =TBE1time
}
one sig SE2 extends NoneStartEvent {}
one sig AT2 extends AbstractTask {}
one sig EE2 extends NoneEndEvent {}
one sig EE1A extends NoneEndEvent {}
one sig EE1B extends NoneEndEvent {}
one sig f1 extends NormalSequentialFlow {} {
  source =SE1
  target =SP1
}
one sig f2 extends NormalSequentialFlow {} {
  source =SE3
  target =AT2
}
one sig f3 extends NormalSequentialFlow {} {
  source =AT2
  target =EE2
}
one sig f4 extends NormalSequentialFlow {} {
  source =SP1
  target =EE1A
}
one sig f5 extends NormalSequentialFlow {} {
  source =TBE1
  target =EE1B
}
one sig Process1 extends Process {} {
  contains =SE1 + SP1 + TBE1 + EE1A + EE1B
}
```

Listing 6.7: *Representation of the Process of Figure 6.3 using Alloy.*

Module ProcessModelCheck.als, groups the list of commands to running the Alloy Analyser w.r.t a set of properties to check. Listing 6.8 shows the commands running the property analysis. As explained in Section 2.6.2.3, there are two ways to operate the analyse with the Alloy Analyser. The *run* command (line 1) allows finding a model that satisfies the formula, while the *check* command (lines 2 to 4) allows finding a counterexample with respect to the formula. Further, it is necessary to specify a bound on each command signatures to limit the search depth. All the bounds are constrained by the process model taken as input to be analysed. For example, if the process model has ten task nodes, the bound of the activity task signature will be 10, *i.e.*, line 1 will be equivalent to [*run CorrectTermination for* 0 *but* 11 *State*, 10 *tasks*]. The keyword **for 0** determines that all the signatures in the analysed model have a bound of 0 by default, excluding for the signatures specified after the keyword **but**. *E.g.*, in Listing 6.8 only the signature of the States number is specified. Thus, the Alloy Analyzer determines the bounds of all the signatures defined in the module *example_TBEI_SP.als* (Listing 6.7) to the default value 0.

```
1      run {CorrectTermination} for 0 but 11 State
2      check {CorrectTermination} for 0 but 20 State
3      check {Safe} for 0 but 15 State expect 0
4      check {SimpleTermination} for 0 but 20 State expect 1
```

Listing 6.8: *Commands for Checking the System.*

6.4.1 Mechanised verification

Alloy comes with a tool, Alloy Analyser, a constraint solver that provides automatic simulation and verification based on a model-finding approach using a SAT solver. Two kinds of verification are available: (1) check the structure itself; (2) check the executions. For the first, *assertions* ensure that the model is well-formed, *e.g.*, a message flow connects two distinct processes. For the second, *predicates on States* are used to express properties on executions. We have defined a set of properties in *PWSProp.als*:

- *Safe*, a predicate that states that no edge or node ever holds more than one token. This property is expressed in Alloy as follows:

```
pred Safe {
  all s:  State, n : Node |s.nodemarks[n] ≤1
  all s:  State, e : Edge |s.edgemarks[e] ≤1
}
```

Listing 6.9: *Safe Property.*

- *SimpleTermination*, a predicate that states that every process reaches a state where an End Event owns a token. This property is expressed in Alloy as follows:

```
pred SimpleTermination {
  all p : Process |some s:  State, n : EndEvent |n in p.contains ∧s.nodemarks[n] ≥1
}
```

Listing 6.10: *Simple Termination Property.*

- *CorrectTermination*, a predicate that states that the whole system reaches a state where all processes have terminated with an End Event and no token is left on other nodes or edges. This property is expressed in Alloy as follows:

```

pred CorrectTermination {
  some s : State |all p : Process |some n:  EndEvent {
    n in p.contains ∧s.nodemarks[n] ≥1
    all nn : p.^contains - n |(nn in EndEvent or s.nodemarks[nn] =0)
    all e : Edge |e.source =p ∧e.target =p  ⟹ s.edgemarks[e] =0
  }
}
```

Listing 6.11: *Correct Termination Property.*

- *EmptyNetTerminationProperty*, extends the *CorrectTermination* with the empty network condition. This property is expressed in Alloy as follows:

```
/* */
pred EmptyNetTermination {
  some s : State {
    all p : Process |some n: EndEvent {
      n in p.contains ∧s.nodemarks[n] ≥1
      all nn : p.contains - n |s.nodemarks[nn] =0
      all e : Edge |e.source =p ∧e.target =p  ⟹ s.edgemarks[e] =0
    }
    all e : MessageFlow |s.edgemarks[e] =0
  }
}
```

Listing 6.12: *EmptyNet Termination Property.*

Other generic properties are available, such as the *MaxTime* predicate that states that the whole system reaches a final state before a given maximal time, and *MinTime* a predicate that states that the whole system takes at least a given minimal time to reach a final state. These properties are expressed with the predicate *NTime*. A counter-example to this predicate means that at least one execution can terminate before max time T. This answers the *MaxTime* property. The validity of this property responds to the *MinTime* correctness property.

```
pred NTime [T : Int]{
some s : State |all p : Process |some n: EndEvent {
      n in p.contains ∧s.nodemarks[n] ≥1
      all nn : p.^contains - n |(nn in EndEvent or s.nodemarks[nn] =0)
      all e : Edge |e.source =p ∧e.target =p  ⟹ s.edgemarks[e] =0
      s.globalclock >T
}
}
```

Listing 6.13: *NTime Property.*

6.5 fbpmn Evaluation

This section presents the evaluation of our contributions, focusing on checking properties on a set of BPMN collaboration and process models.

6.5.1 Experiments using the TLA$^+$ Encoding/Tooling

Experiments were conducted on a laptop with a 1.9 GHz (turbo 4.8 GHz) Intel Core i7 processor (quad-core) with 32 GB of memory. Results are presented in Table 6.3. The first column is the reference of the example in our archive at `https://github.com/pascalpoizat/fbpmn/` under */models/bpmn-origin/src* folder. The characteristics of a model are the number of participants, the number of nodes (incl. gateways), the number of flow edges (sequence or message flows), whether or not the model is well-balanced (for each gateway with n diverging branches, we have a corresponding gateway with n converging branches) and whether or not it includes a loop. The communication models are asynchronous (bag), FIFO-ordered between each couple of processes (FIFO pair), globally FIFO (FIFO all), or synchronous-like (RSC). The results of the verification then follow. First, data on the resulting transition system are given: number of states, number of transitions, and the depth (the length of the longest sequence of transitions that the model checker has to explore). For each of the three correctness properties presented above, we indicate if the model satisfies it. Lastly, the accumulated time for the verification of the three properties is given. Our tool supports more verifications (see Table 7.1) and can be easily extended with new properties. We selected these three ones since they are more BPMN specific [23].

Table 6.3 presents the results for a selection of properties and models from our archive montionned above, for a variety of gateways and activities. These illustrative examples include realistic business process models (001 and 002 two client-supplier models, 040 from Figure 2.8, 017 from [108], and 020 from [138]), and models dedicated to specific concerns: termination end events and sub processes (007–011 from [23]), inclusive gateways (003, 012, 013 and 018), exclusive and event-based gateways (015 and 016).

A first conclusion is that verification is rather fast: the verification of one property generally takes just a few seconds per model, the longest being for model 020 that takes up to 53s of accumulated time for the three properties (5s for the construction of the state space). Experiments also show the effect of

Table 6.3: *Experimental Results.*

ref.	Characteristics					Com.	LTS size			validity			total
	proc.	nodes (gw.)	SF/MF	B	L	model	states	trans.	depth	(6.1)	(6.2)	(6.3)	time
001	2	17 (2)	14/3	✓	×	bag	93	173	25	✓	✓	✓	3.60s
						fifo pair	85	161	21	✓	×	×	3.26s
						RSC	77	147	19	✓	×	×	3.66s
002	2	16 (2)	13/3	✓	×	bag	79	147	23	✓	✓	✓	3.57s
						fifo pair	71	135	19	✓	×	×	3.61s
						RSC	63	121	17	✓	×	×	3.56s
003	1	14 (6)	16/0	×	✓	none	41	59	15	✓	✓	✓	3.10s
006	2	20 (4)	18/5	×	✓	bag	470	966	43	×	×	✓	4.20s
						fifo all	522	932	40	×	×	×	4.87s
						RSC	247	420	38	×	×	×	4.13s
007	1	8 (2)	7/0	×	×	none	44	73	15	×	×	×	2.52s
008	1	11 (2)	9/0	×	×	none	48	77	19	×	✓	✓	2.70s
009	2	12 (2)	9/1	×	×	bag	170	395	19	×	×	×	3.70s
010	2	15 (2)	11/1	×	×	bag	186	423	23	×	×	✓	3.72s
011	2	15 (2)	11/1	×	×	bag	100	209	21	×	✓	✓	3.51s
012	1	15 (8)	17/0	✓	✓	none	71	137	15	✓	✓	✓	3.85s
013	1	17 (8)	21/0	✓	✓	none	407	1049	15	✓	✓	✓	5.93s
018	1	19 (8)	25/0	✓	✓	none	4631	15513	18	✓	✓	✓	30.23s
015	2	14 (2)	10/2	×	×	bag	68	117	11	✓	×	×	3.11s
016	2	14 (2)	10/2	×	×	bag	36	53	11	✓	✓	✓	3.17s
017	1	32 (12)	36/0	×	×	none	93	141	37	✓	✓	✓	4.03s
020	4	39 (6)	34/8	×	✓	bag	3558	11035	52	✓	✓	✓	20.76s
020						fifo all	2138	5654	52	✓	✓	✓	14.57s
020						RSC	1030	2695	52	✓	×	×	10.26s
040	2	29 (3)	23/7	✓	✓	bag	353	712	38	✓	×	✓	6.26s
040						fifo	297	616	35	✓	×	×	6.22s
040						ad-hoc p.131	657	1154	39	✓	×	✓	9.5s

the communication model on property satisfaction (models 001, 002, 006, 020), the use of TLA$^+$ fairness to avoid infinite loops (012, 013, 018, 020), and the use of terminate end events combined with model constraints (see Section 6.6.4.1) to deal with unsafety (006).

LTL verification is $O(M * 2^F)$, where M is the size of the state space, and F is the size (expressed in terms of the number of involved temporal operators) of the formula. F is mainly influenced by the number of fairness constraints. Regarding M, in practice, more than the size of the BPMN schema, interleaving is the main cause of state explosion. Interleaving is directly linked to the number of processes. Thus, more than the number of nodes (which has a limited impact), the verification time is mainly impacted by the number of processes and their branching.

6.5.2 Experiments using the Alloy Encoding/Tooling

Experiments were conducted on a laptop with a 3.9 GHz (turbo 3.30 GHz) Intel Core i5 processor (quad-core) with 64 GB of memory. First, the processes and their properties are translated into an Alloy specification using fbpmn translator to perform the experiments. Then, this specification is given as input to the Alloy Analyser, which reduces the verification to an SAT problem. Therefore, the Alloy Analyser presents the specification to a SAT solver in CNF format. A CNF is a conjunction of clauses. Each clause is a disjunction of variables (cf. Section 2.4.11). A satisfying assignment of a SAT problem consists of a Boolean assignment to the variables such that all clauses are satisfied. It is usual to use the number of variables and clauses as a measure for a SAT problem complexity. The SAT solver used in the following is MiniSat (one of those supplied by default with the Alloy Analyser).

Results are presented in Table 6.4. The first column is the reference of the example in our archive at `https://github.com/pascalpoizat/fbpmn/` under */AlloyTest/TLARepresentation/* folder. Column 2 represents the variant of the commands used (run or check). Column 2 exhibits the analysed property from Section 6.4. Column 3 depicts the limit state number used for each checked property. Columns 4 and 5 illustrate the number of variables and clauses, respectively. Columns 6 and 7 show the time to generate the CNF and to solve the SAT problem, respectively. Finally, column 8 specifies the result of the verification. Note that when there is communication, we use a bag communication model. The verification results are depicted as follows: First, if there is any counter-example, we display (CE). If an instance is found when applying the run command, we display (Instance). Lastly, If neither of these two results is available and the assertion may be valid, as expected, we show the (✓) mark. However, if the assertion is inconsistent, we display the ($\times$) mark.

These results highlight the effectiveness of our tool w.r.t. a set of concrete models from our repository. A first conclusion is that verification is relatively fast: the solving time of one property generally takes just a few seconds per model, the longest being for model 001 that takes up to 8s even if the whole generated SAT problems present a relatively high complexity (over 14 thousand variables and over 1 million clauses).

Secondly, checking properties provides promising results since we find the results are consistent with the expected ones. More precisely, we applied different properties for each model. For example, in the model (001), which represents a collaboration, we have used the checking of the *EmptyNetTermination* property to ensure that the results are consistent with that obtained in the previous experiment (cf. Line 1, Table 6.3) using the TLA$^+$ translation. On the other hand, the model shows that the correct termination property is unsatisfied using 9 states and generates a counterexample. However, this property is satisfied for the same model using 25 states. This latter is due to the Alloy Analyser feature. The Alloy Analyser is based on a SAT solver that performs Bounded Model Checking (BMC). Thus, it is only able to guarantee the absence of counterexample up to some bound k. Consequently, the Alloy Analyser cannot determine, on its own, the total number of states needed to analyse the model entirely. It is up to the user to choose this bound number and that it must be greater than the number of States typically required to unfold the transition relationship completely.

Model (t002) presents a simple BPMN process that contains an *Intermediate Timer Catch Event* with a duration value equal to 4. Therefore, the *CorrectTermination* must be checked with more than nine states to give enough time for the globalclock / localclock to reach 4. Line 10 (the fourth checked property of this model) defines an empty run. This last produces a random instance of the model satisfying the facts for guaranteeing at least one model execution instance (no deadlock).

In model (t003), Line 16 shows an Instance producing of the non-simultaneous reachability of the end events (EE1A) and (EE1B) formula means that the *Interrupting Timer Intermediate Event* behaves as expected. However, running this formula for the model (t004) fails to find an instance (*i.e.*, the set of clauses for which no satisfactory instance exists). The latter means that the *non-interrupting Timer*

Table 6.4: *Metrics from the Alloy Analyser Executed on a Subset of Examples.*

Ref.	Command	Property	States Bound	Variables	Clauses	CNF (s)	SAT (s)	Result
001	Check	Safe	15	9201	428803	219.12	2.55	✓
		Simple Termination	25	14923	1256715	372.097	8.93	✓
		Correct Termination	25	14971	1284401	370.04	3.96	✓
		Correct Termination	9	5787	436257	122.89	0.2	CE
		Empty Net Termination	15	14971	1263643	373.0405	4.24	✓
	Run	Safe	11	6913	569985	154.377	0.51	Instance
t002	Check	Safe	10	10389	105806	5.328	0.015	✓
		Simple Termination	10	1318	78861	0.83	0.003	✓
		Correct Termination	10	1327	79548	0.693	0.003	✓
	run	{ }	8	1061	61215	0498	0.02	Instance
t003	Check	Safe	10	2689	222050	4.82	0.183	✓
		Simple Termination	15	3970	317547	7.10	0.16	✓
		Correct Termination	15	3984	320475	6.929	0.113	✓
	Run	EE1A Reachability	9	2442	182379	3.927	0.003	Instance
		EE1B Reachability	11	2956	226939	4.926	0.595	Instance
		Non-simultaneous Reachability of EE1A and EE1B	20	20657	2049285	27.957	0.595	Instance
t004	Check	Safe	10	1832	133399	1.577	0.027	✓
		Correct Termination	15	2732	186827	2.32	0.027	✓
	Run	EE1A reachable	7	1308	79861	0.917	0.009	Instance
		EE1B reachable	9	1664	105979	1.228		Instance
		Non-simultaneous Reachability of EE1A and EE1B	14	2540	170826	2.019	0.035	×
t005	Check	Safe	10	3616	331101	36.887	6.436	✓
		SimpleTermination	14	4978	431421	41.620	9.101	✓
		CorrectTermination	14	5004	437717	48.570	8.980	✓
		EmptyNetTermination	14	4976	433821	42.218	0.963	CE
	Run	! EmptyNetTermination	10	3616	304741	33.814	0.564	Instance
		EmptyNetTermination	10	3632	310097	30.646	1.539	Instance
		EE2A Reachability	9	3285	269261	23.492	0.590	Instance
		EE2B Reachability	9	3285	269261	21.967	0.206	Instance

Intermediate Event behaves as expected.

Finally, for model (t005), Line 25 shows that checking the EmptyNetTermination property produces a counterexample. The latter means that it is invalid by taking the Timer Intermediate Catch Event Branch. For that, both non-EmptyNetTermination and EmptyNetTermination running properties produce an Instance as both cases are reachable (*i.e.*, the first instance means that the formula is UnSAT and there is a possibility of executing the receiving branch may be taking. But, the second one means that the timer event may be fired before the message receiving.

It is important to note that if no counterexample is found when checking a strong property (check), checking its weak equivalent will certainly find at least one instance that satisfies the property. For example, suppose no counterexample is found to go into a no safeness situation (check Safe, cf Line 1). In that case, obtain an instance such that the process safe is trivially true (run Safe, cf Line 6). Indeed, in general, to generate counterexamples during verification (check F), the Alloy Analyser attempts to find instances of the negation of the formula (*i.e.*, $[run\ F \ \equiv\ check\ \neg\ F]$). Thus, as a general rule, it is interesting to evaluate a weak property only when the strong property has returned a counterexample.

Current experiments have allowed us to validate our semantics on a subset of study cases models, and the implementation proved the feasibility of our approach, but unfortunately, real-life models are often out of reach of Alloy Analyzer as the number of required states for an execution exceeds its capacity. Still, verification is achieved in a reasonable time.

6.6 The fbpmn Supporting Tool

This section presents the open-source software made available at `https://github.com/pascalpoizat/fbpmn`, that can be redistributed and eventually modified under the terms of the GPL2 License. We here choose to focus on parts related to the TLA$^+$ implementation of the semantics given in Chapter 4, verification using TLC, and the associated Desktop usage and Web application. These parts being more polished than the ones related to Alloy and Space BPMN support [205].

6.6.1 Architecture and General Principles

fbpmn tool suite for TLA$^+$, is made up of :

- the fbpmn program and several accompanying scripts to perform verification in a single command line and to graphically animate counterexamples.
- a Web application version of the above, with a client-side front-end (for BPMN modelling and for giving communication and verification parameters) that runs in a single browser, and a server-side back-end verification engine, built around fbpmn and scripts, for which a Docker version is available

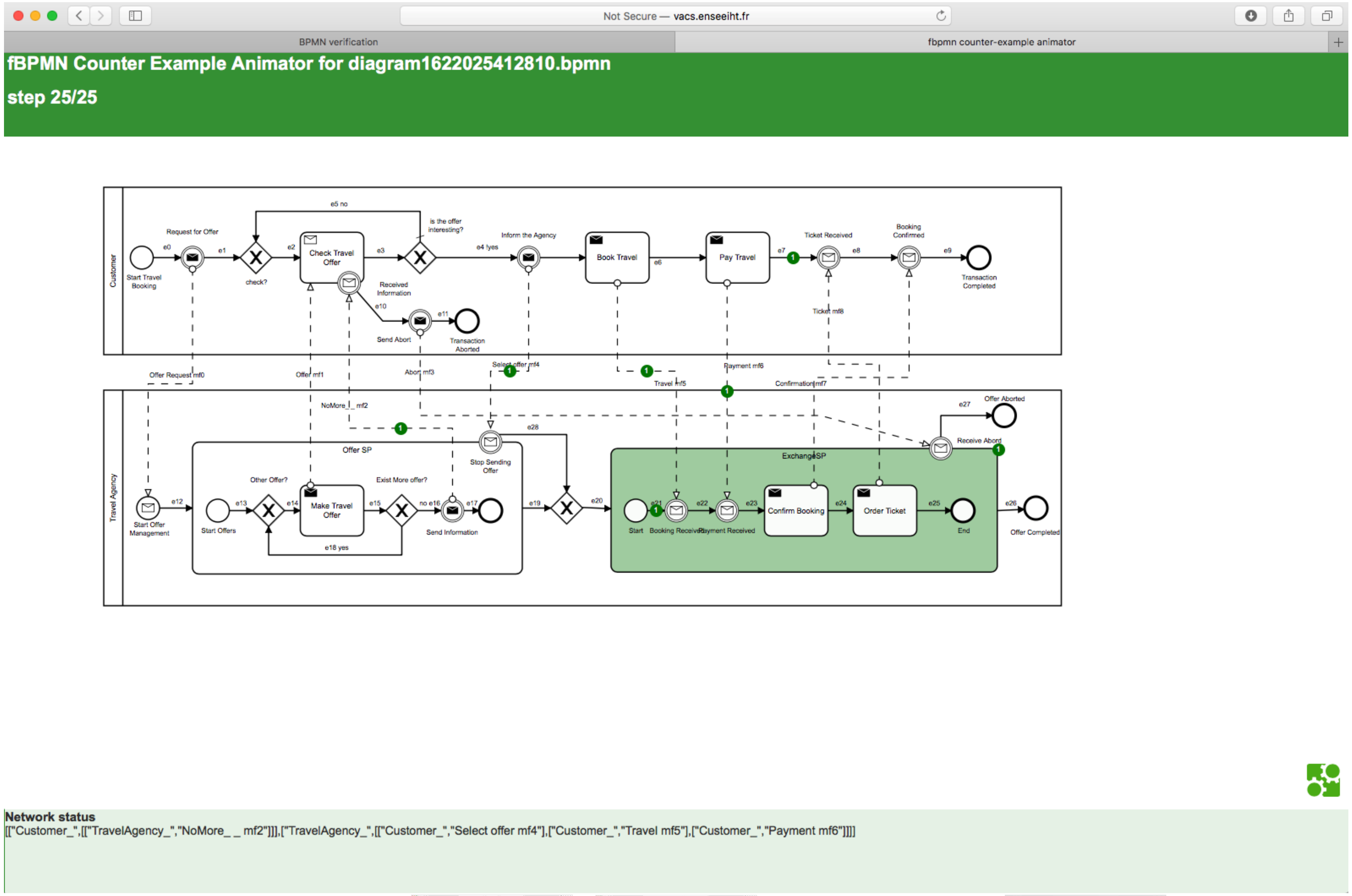
The fbpmn tool suite is centred around a command, FBPMN, that is available for Linux, OSX, and Windows (binaries are available for the first two, the latter requiring, for now, a compilation process). This command is used to transform a BPMN model into TLA$^+$ representation. fbpmn is also in charge of the computation of the Pre_N and Pre_E sets that are used in the semantics of the OR gateways since these two sets can be structurally computed from the BPMN graph structure. This generated TLA$^+$ graph module is then passed, together with modules for TLA$^+$ implementation of our well-formedness rules and semantics, to the TLC model checker, as described in the bottom of Figure 6.1.

In the case where a verification fails, TLC outputs a counter-example as a state trace that includes for each step, the state of the markings and the communication network (Definition. 4.4.1). To ease the interpretation of this by the process designer, fbpmn can also be used to generate an interactive animation of the counter-example, where one can see the marking over the BPMN model and navigate between the steps of the counterexample (Figure 6.5). The presentation layer for the counterexample animator has been achieved using the Camunda.io javascript library[2].

6.6.2 Desktop Modelling and Verification

For a given model, one may have different properties of interest (*e.g.*, safety, soundness, and message-relaxed soundness), and since several communication models are possible (*e.g.*, the seven ones presented in Table. 4.3), it would be tedious to run fbpmn for each of the combinations. Hence, we provide the process

[2]bpmn-js: https://github.com/bpmn-io/bpmn-js



Not Secure — vacs.enseeiht.fr

BPMN verification | fbpmn counter-example animator

fBPMN Counter Example Animator for diagram1622025412810.bpmn

step 25/25

Network status
[["Customer_",[["TravelAgency_","NoMore_ _ mf2"]]],["TravelAgency_",[["Customer_","Select offer mf4"],["Customer_","Travel mf5"],["Customer_","Payment mf6"]]]]

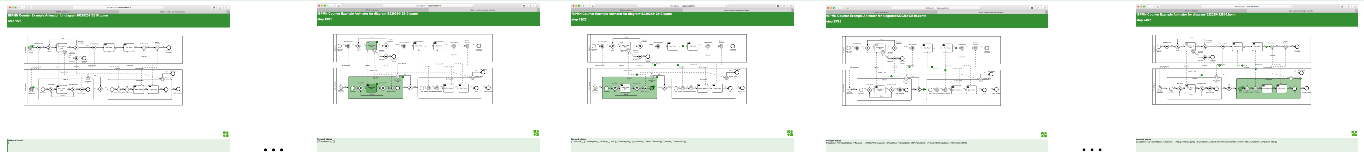
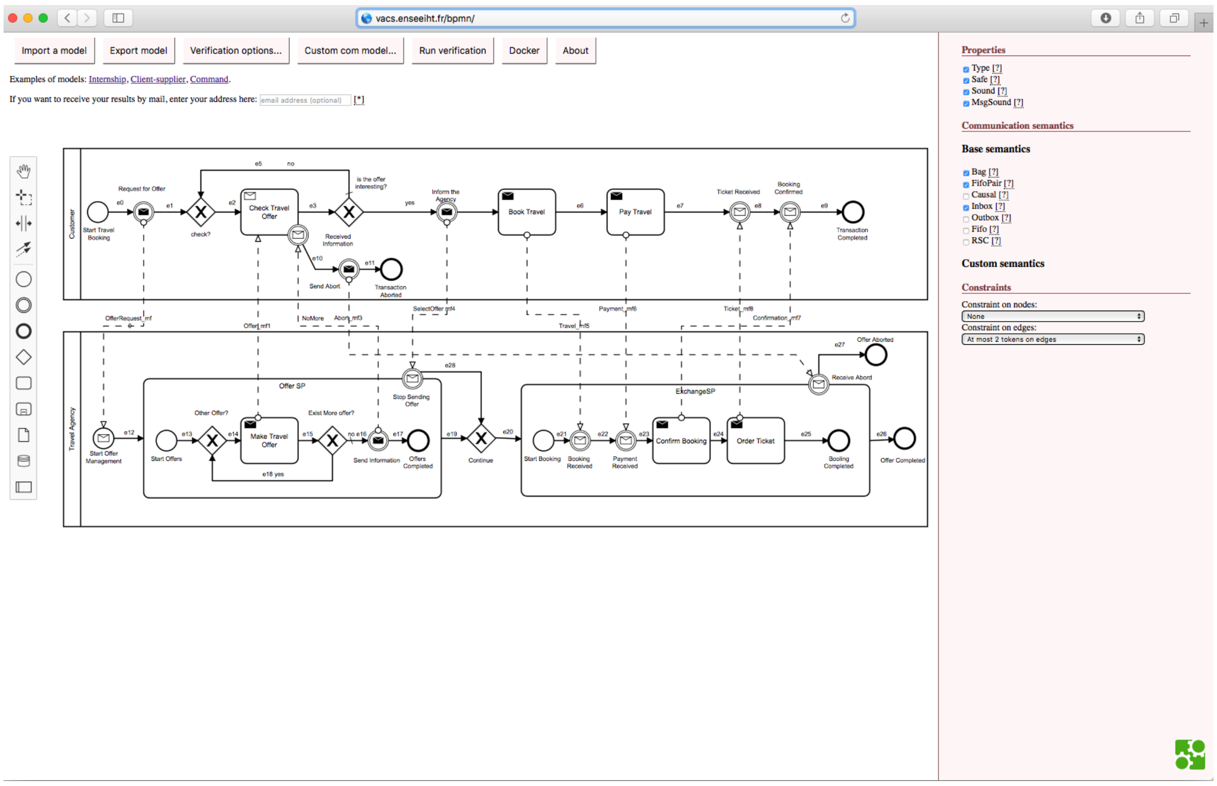


Table 6.5: *Animation of a Counter-example (Model in Figure 2.8, for Soundness with Fifo Inbox Mode) Generated with* fbpmn.



Examples of models: Internship, Client-supplier, Command.

If you want to receive your results by mail, enter your address here: email address (optional) [*]

Properties

- ☑ Type [?]
- ☑ Safe [?]
- ☑ Sound [?]
- ☑ MsgSound [?]

Communication semantics

Base semantics

- ☑ Bag [?]
- ☑ FifoPair [?]
- ☐ Causal [?]
- ☑ Inbox [?]
- ☐ Outbox [?]
- ☐ Fifo [?]
- ☐ RSC [?]

Custom semantics

Constraints

Constraint on nodes: None

Constraint on edges: At most 2 tokens on edges

Figure 6.4: fbpmn *Web Application (modelling and verification panel).*

designer with scripts (only under Linux or OSX) that ease verification. When the designer launches the fbpmn-check script, it reads a configuration directory and runs fbpmn based on the designer preferences. Let us suppose the configuration directory is as follows.

```
Network01Bag.tla      Network04Inbox.tla    Network07RSC.tla      Prop03Sound.cfg
Network02FifoPair.tla Network05Outbox.tla   Prop01Type.cfg        Prop04MsgSound.cfg
Network03Causal.tla   Network06Fifo.tla     Prop02Safe.cfg
```

This will yield four different properties to be checked for seven different network models, generating at most 28 counter-example traces. Running the fbpmn-logs2html script on a working directory generated by fbpmn-check, finds out these counter-examples and generates an interactive animator for each of them. It is also possible to give fbpmn-check a number of cores to use, and this value is passed to the TLC model checker.

6.6.3 Online Modelling and Verification

To ease the use of the fbpmn tool suite, we have implemented a Web application for it (Figure 6.4).

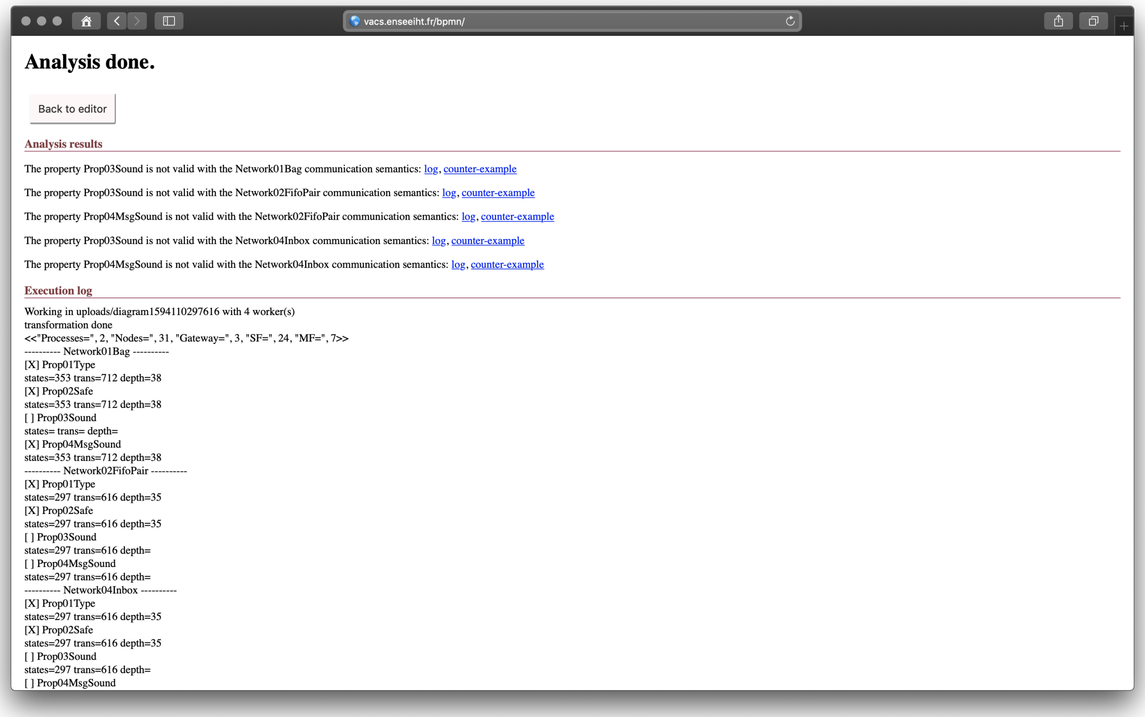
There, the user can import, design, or export a BPMN model (this is achieved using the Camunda.io framework). Then verification parameters can be given: which properties to check, which communication models to check with, possibly model constraints (see below) for nodes and/or edges.

After retrieving the results (Figure 6.5), the user can see a textual version of counter-examples and/or animate it on the model as presented in Section 6.6.1.

The fbpmn Web application is available online at [206] for demonstration purposes. Yet, if one is interested in it, we advocate its deployment on one's own machine or server. For this, we provide a Docker image, downloadable from our Web application at `https://github.com/pascalpoizat/fbpmn/tree/master/web`.

6.6.4 Extensibility

Our framework can be extended as far as safeness constraints, properties to check, and communication models are concerned.



Analysis done.

Analysis results

The property Prop03Sound is not valid with the Network01Bag communication semantics: log, counter-example

The property Prop03Sound is not valid with the Network02FifoPair communication semantics: log, counter-example

The property Prop04MsgSound is not valid with the Network02FifoPair communication semantics: log, counter-example

The property Prop03Sound is not valid with the Network04Inbox communication semantics: log, counter-example

The property Prop04MsgSound is not valid with the Network04Inbox communication semantics: log, counter-example

Execution log

```
Working in uploads/diagram1594110297616 with 4 worker(s)
transformation done
<<"Processes=", 2, "Nodes=", 31, "Gateway=", 3, "SF=", 24, "MF=", 7>>
---------- Network01Bag ----------
[X] Prop01Type
states=353 trans=712 depth=38
[X] Prop02Safe
states=353 trans=712 depth=38
[ ] Prop03Sound
states= trans= depth=
[X] Prop04MsgSound
states=353 trans=712 depth=38
---------- Network02FifoPair ----------
[X] Prop01Type
states=297 trans=616 depth=35
[X] Prop02Safe
states=297 trans=616 depth=35
[ ] Prop03Sound
states=297 trans=616 depth=
[ ] Prop04MsgSound
states=297 trans=616 depth=
---------- Network04Inbox ----------
[X] Prop01Type
states=297 trans=616 depth=35
[X] Prop02Safe
states=297 trans=616 depth=35
[ ] Prop03Sound
states=297 trans=616 depth=
[ ] Prop04MsgSound
```

Figure 6.5: fbpmn *Web Application (verification results).*

6.6.4.1 Model Constraints

Some models are unsafe, *i.e.*, the semantics can yield an infinite marking on some node(s) or edge(s). In such a case, one may rely on model constraints associated with the BPMN model to be verified. Given the model is model.bpmn, one just has to create a file model.constraint of the form:

```
CONSTANT ConstraintNode <- <ConstraintOnNodes>
         ConstraintEdge <- <ConstraintOnEdges>
         Constraint <- <Overall constraint in terms of ConstraintNode and ConstraintEdge>
```

Some node constraints and edge constraints are already defined in our TLA$^+$ library, *e.g.*, the one to state that an edge should have at most two tokens on it, MaxEdgeMarking2, or the one to limit the number of tokens only on message edges MaxMessageEdgeMarking2. The most usual constraint combinator is also already defined there, ConstraintNodeEdge, which is the conjunction of the user-specified node and edge constraints. Using this, we may verify model 006 (as seen in Section 6.5), defined in file e006TravelAgency.bpmn, with a file e006TravelAgency.constraint:

```
CONSTANT ConstraintNode <- TRUE
         ConstraintEdge <- MaxEdgeMarking2
         Constraint <- ConstraintNodeEdge
```

The user is free to extend our constraint library by extending the PWSConstraints.tla TLA$^+$ module.

6.6.4.2 New Properties

We support several properties from the literature. However, it is possible to extend this set. To do so, one has to:

1. Define a new property, say MyProperty, at the end of the main TLA$^+$ semantic module, PWSSemantics.tla;
2. Create a new file PropNNMyProperty.cfg in the fbpmn configuration directory, with NN being a number different from the existing properties there;

3. In the contents of PropNNMyProperty.cfg refer to the property name given in 1.

The definition of new properties has some limitations. First, these properties must be defined using LTL since this is the logic that is checked by TLC. Second, these properties must cope with our definition of state (Definition 4.4.1), *i.e.*, they can be defined in terms of node markings, edge markings, and/or network markings. Properties can also refer to the types of the nodes and edges, as shown in Section 6.3.2 for the soundness property.

6.6.4.3 New Communication Models

As stated before, we support the most usual communication models to be used as parameters for the BPMN semantics. Still, one may define new models. To achieve this, one has to:

1. Define the new communication model semantics, say MyNet, in a NetworkMyNet.tla file in the fbpmn TLA$^+$ theories directory;
2. Copy one of the files in the fbpmn configuration directory to a new file NetworkNNMyNet.tla in the same directory, with NN being a number different from the existing communication models there;
3. In the contents of NetworkNNMyNet.tla changes the line of the network implementation definition to refer to the new communication model as defined in 1.

6.7 Summary

In this chapter, we have presented our fbpmn tool. We have detailed its architecture, features, verification mechanism and evaluated its practicability over a set of examples. The evaluation section demonstrates the achievement of the objectives defined in the introduction. The BPMN formalisation, its implementation, and the associated tool make it possible to verify all perspectives of business processes automatically.

To sum up, the tool proved the feasibility of our approach. However, some improvements are already under realisation (cf. 7.4). The next chapter concludes this thesis by summarising our contributions, presenting its limitation, and giving some perspectives.

Part IV

Conclusion and Future Work

Chapter

7 Conclusion

In this chapter, we recall the objectives of the thesis in Section 7.1. We present the contributions of the work we carried out in Section 7.2, then we position this work in relation to the literature in Section 7.3. Finally, we present our perspectives in Section. 7.4.

7.1 Objectives Remainder

The main objectives of this thesis were: (1) to provide a direct formal execution semantics for a subset of BPMN elements that supports sub-processes, communication, and time constructs and is parametric with reference to the properties of the communication; and (2) to support this formalisation with tools that automatically perform the verification of correctness properties for BPMN collaboration models. Our work, therefore, aimed at improving and facilitating the process of formal specification and verification of the business process models, which is a long and complicated task and requires a knowledge of formal tools in order to avoid an in-depth review of the code and the specification in failure after implementation.

7.2 Contributions

This thesis contributes to our objectives with the *fbpmn framework* enabling the modelling, the analysis and the formal specification of collaborations and time-aware models based on a well-founded set of formal semantics rules. In the following, we summarise our contributions by answering to the thesis research questions presented in Chapter 1, Section 1.2, which were:

- Q1. Does the correctness of BPMN collaboration diagrams depend on the used communication models?
- Q2. How to precisely describe the formal semantics of BPMN collaboration diagrams, taking into account different communication models?
- Q3. How to formalise the execution semantics of the BPMN time constructs, including their relation to the ISO-8601 standard format?
- Q4. What are the time process patterns supported by the BPMN standard, and does our semantics support all of them?
- Q5. How to verify such formal models?, which are the properties of interest?, and can the formal semantics of the BPMN collaborations drive the development of software tools based on BPMN collaboration diagrams?

Answering Q1 and Q2. Firstly, we have defined a direct formalisation for BPMN collaboration diagram elements. We use First-Order Logic (FOL) with natural numbers, sets, and maps. Instead of using a formal intermediary model, *e.g.*, Petri nets or a process algebra, this choice of a simple yet expressive framework enables one to get a formal semantics that is amenable to implementation in different formal frameworks while still being close in its structured presentation to the semiformal semantics of the standard (hence it can be related to it). As far as the subset of BPMN is concerned, we have first included the generic control flow elements (*i.e.*, gateways, tasks, and events). We have then taken constructs with complex execution semantics into account, mainly relative to our focus: creation and termination of processes based on messages or time, message and time-related intermediary events and boundary events (interrupting or non interrupting), event-based gateways, inclusive-join gateways, and subprocesses. Secondly, the provided semantics is parametric regarding the properties of the communication model. Therefore, we support seven point-to-point communication models relating to the message-passing behaviours between and within processes and define their formal semantics. These communication models are important since, as seen for example in Table 6.3, the chosen model has an impact on whether the correctness properties of a BPMN model are fulfilled or not.

Answering Q3 and Q4. Firstly, we have extended our formal semantics to support the time-related constructs of BPMN. In the first, time semantics was abstracted in a non-deterministic way. This new semantics supports different combinations of events, time information categories (timeDate, timeDuration, and timeCycle) and the corresponding ISO-8601, descriptions as prescribed by the BPMN standard. As seen in Table 5.2, we support 13/31 time semantics features in BPMN. Secondly, we have formalised the execution semantics of a set of time patterns specified in BPMN. Many authors have underlined the importance of time patterns and a lot of effort has been made to identify the most common time-related scenarios from a business perspective, namely, *Process Time Patterns* [207]. These patterns were defined only in terms of textual descriptions for PAIS system in general. Our work was first to provide a graphical description of these patterns to assess the suitability of BPMN to express these common time-related scenarios. Then, to validate the provided semantics for the time constructs given in Chapter 5, as we show its suitability to cover the process time patterns expressed in BPMN. Roughly speaking, our work demonstrated that our semantics supports 7/10 from ten of these patterns.

Answering Q5. Based on the proposed FOL semantics, this thesis provides an automatic verification tool-suit, called fbpmn, for the business process collaboration models. fbpmn is based on the translation of the FOL semantics into the TLA$^+$ and Alloy languages and the use of the *TLC* model checker and the Alloy Analyser for the model analysis. Thus, within the fbpmn tool both standard model checking and bounded model checking techniques are integrated to effectively support verification. As far as properties of interest are concerned, fbpmn tool allows checking domain-specific properties dedicated either to workflow notations in general (*e.g.*, soundness, safety, and deadlock-freedom) or to BPMN in particular (*e.g.*, simple termination, correct termination, message relaxed soundness) and it allows animating counter-examples to fix erroneous models in case of checking properties fails. Further, fbpmn tool is proposed either as a desktop or as a Web application to model, check, and correct the business collaboration models. The latter provides transparency to the users with reference to the formal background.

7.3 Position with Reference to the Litterature

This section compares the most relevant attempts to formalise the semantics of BPMN, which cover the interaction and time characteristics with the work at hand.

7.3.1 Collaboration-Based Approaches

As shown in Chapter 3, numerous works in the literature have focused on the formalisation of BPMN and on the verification support for the collaboration diagrams and communication features in BPMN [23, 71, 86, 89, 125, 128, 132, 138]. We add [64] due to its role as a seminal paper and [95, 117] due to their representatives formal model they use. Table 7.1 gives a synthetic presentation of a comparison between these proposals and ours. The table focuses on (1) BPMN features, and (2) properties of interest that are supported by each work. This table divides the approaches between those that rely on an intermediary model and those that have the benefit of providing a direct link between BPMN constructs and the verification formalism. Our work follows this line. Further, our choice of FOL lets us implement the semantics in different tools *e.g.*, TLA$^+$ and Alloy as here or SMT solvers for the future. As far as the BPMN coverage criteria are concerned, we can observe that we are among the approaches with high coverage. To make verification tractable, we have abstracted from the data and the multi-instance constructs, that are often related to data. Most of the work, still, support the verification of business process correctness properties or, at least, all-purpose formal properties (reachability, deadlock). To the best of our knowledge, these approaches do not support verification of BPMN models under a specific, and parametric, communication model.

7.3.2 Time-Based Approaches

As shown in Chapter 3, numerous works in the literature have focused on the formalisation of BPMN time-related models constructs. Among these works we select [81, 99, 104, 116, 119, 142, 143, 208] for

Table 7.1: *Comparison of Tool-Supported Approaches for the Analysis of Communication in BPMN.*

Approach		Transformation						Direct					
Reference		[89]	[86]	[71]	[117]	[95]	[64]	[138]	[23]	[132]	[125]	[128]	**ours**
Year		2018	2017	2013	2011	2008	2008	2018			2012	2014	2020
Formalism		CPN	ECATNets	In-Place Graph	CSP	YAWL Nets	PN	LTS			LTL	Maude	FOL
Tool		not avail. CP4PBMN	not avail. BPMNChecker	yes GrGen	yes Machine-readable CSP	yes BPMN2YAWL	yes transformer	yes MIDA	– –	yes BProVe	– –	not avail. prototype	yes fBPMN
Supported Elements	*AND* gateway	●	●	●	●	●	●	●	●	●	●	●	●
	XOR gateway	●	●	●	●	●	●	●	●	●	●	●	●
	OR gateway	●	–	●	●	●	●	●	●	●	●	●	●
	EB gateway	●	●	●	–	●	●	●	●	●	●	–	●
	ST	●	●	●	●	●	●	●	●	●	●	●	●
	RT	●	●	●	●	●	●	●	●	●	●	●	●
	MSE	●	●	●	–	●	●	●	●	●	●	–	●
	TMIE	●	–	●	–	–	●	●	●	●	●	–	●
	CMIE	●	–	●	–	–	●	●	●	●	●	–	●
	MBE	–	–	–	●	–	–	–	–	–	●	–	●
	TBE	–	–	–	●	●	–	–	–	–	●	–	●
	TICE	–	●	–	●	●	●	–	–	–	●	–	●
	TSE	–	–	–	–	●	–	–	–	–	●	–	●
	MEE	●	●	●	–	–	–	●	●	●	●	–	●
	TEE	–	–	●	●	–	–	●	●	●	●	–	●
	sub-processes	●	●	●	–	●	●	●	–	–	●	●	●
	data	●	●	●	–	–	–	●	–	–	–	●	–
	multi instance (pools)	–	–	●	–	–	–	●	–	–	–	–	–
	multi instance(activities)	●	●	●	●	●	●	●	–	–	–	–	–
Verification	option to complete	–	●	–	–	●	–	–	–	●	–	●	●
	proper completion	–	–	–	–	●	●	–	–	●	–	●	●
	no dead activity	–	–	–	–	●	●	–	–	●	–	●	●
	safety	–	–	–	●	–	–	–	●	●	●	–	●
	collaboration soundness	–	–	–	–	–	–	–	●	–	–	–	●
	msg relaxed soundness	–	–	–	–	–	–	–	●	–	–	–	●
	undelivered messages	–	–	–	–	–	–	–	–	–	–	–	●
	general-purpose	●	●	●	●	●	●	–	●	●	●	●	●
Communication models		–	–	–	–	–	–	–	–	–	–	–	●

the diversity of the formal representation used. We add [209] as it is the referenced work that formalises all the process time patterns in PAIS. Table 7.1 gives a synthetic comparison between these proposals and ours. The table focuses on (1) covering all the time events in BPMN, considering their categories, and (2) showing how these works support the presentation of time patterns. As far as the BPMN coverage criteria are concerned, we can observe that we are among the approaches with high coverage. A few studies address the evaluation of BPMN expressiveness with respect to its modelling elements, and most of them extend the notation in order to enhance the support of the standard towards time management constraints [99, 116, 119, 142, 143, 208]. As highlighted in Table 7.1, most of the existing works treat time duration for activities extending BPMN by: (1) defining a non-deterministic delay for a task [116] and [99] or, (2) representing a fixed duration a specified as an $[a, a]$ interval [81]. However, BPMN gives the possibility to represent a duration for activity using its own elements, without any extensions (see Chapter 5). In addition, and to the best of our knowledge, no work in the literature allows one to specify the semantics for the different types of time information (*i.e.*, timeDate, timeCycle, timeDuration) associated with BPMN time-related events. In this thesis, we cover the defined set of BPMN timer events in their full generality. As an example, consider the timer boundary event with cycle type. BPMN defines the cycle type with reference to ISO standard definition, where the ISO cycle type definition represents a complex construct which may be a repetition based on a duration until date or a repetition defined by a starting date and a period, or others (Table 5.2). To the best of our knowledge, most papers do not support all the variations of this construct (see Table 7.2).

However, some works [210], limited by the absence of a formalization, propose a simplified version of these events, *e.g.* every 10 minutes (a repetition on a defined period). Note that, even if the work in [209] provides a very rich formal semantics for time-related process patterns in terms of temporal execution trace, it is given for PAIS systems in general and does not address their semantics with reference to the semantics of the BPMN time constructs. In addition, the proposed semantics does not enable the verification and does not show their coverage *w.r.t.* standard BPMN elements.

Table 7.2: *Comparison between Approaches Supporting BPMN Time-Constructs.*

	Reference	[116]	[208] [119]	[99]	[14] [104]	[209]	[81]	[142] [143]	**ours**
	Year	2008	2010-2012	2011	2013-2014	2016	2017-2019	2017-2018	2020
	Formalism	CSP	CSP+T	TA		Timed Ex. Traces	Timed PN	Maude	FOL
BPMN	TSE	●	●	–	–	–	–	●	●
	$TICE$	●	●	●	●	–	●	●	●
	TBE non-interrupt	–	●	–	–	–	–	●	●
	TBE interrupt	●	●	●	●	–	–	●	●
Time	timeDate	–	–	–	–	●	–	–	●
	timeCycle	–	–	–	–	–	–	–	●
	timeDuration	●	●	●	●	●	●	●	●
Patterns	time lag between activities	●	●	–	●	●	–	●	●
	duration	●	●	–	–	●	●	–	●
	time lags between arbitrary events	–	–	–	–	●	–	–	●
	fixed date element	–	–	–	–	●	–	–	●
	shedule restricted element	–	–	–	–	●	–	–	–
	time based restriction	–	–	–	–	●	–	–	–
	validity period	–	–	–	●	●	–	–	–
	time dependent variability	–	●	–	–	●	–	–	●
	cycle element	–	–	–	–	●	–	–	●
	periodicity	–	–	–	–	●	–	–	●
Other	time duration for activities	●	●	●	●	–	●	●	●
	time interval for edges	–	●	–	●	–	–	●	–

7.4 Limitations & Perspectives

Formal semantics support for BPMN models is an exhaustive research area, which can be only partially covered by one thesis. Therefore, our work is subject to some limitations. Some of them were already mentioned in the summary section of each chapter. In this section, we discuss these limitations, and we give some ideas for dealing with them.

Some features that play a role in full-fledged executable collaborations have been discarded here. This is the case of the data (data objects, data stores, assignments, and message payloads) and multi-instance (for activities and pool lanes) constructs.

- **Data constructs support.** To deal with data, a direct (and usual) solution is to extend the notion of state with a substitution from variables to values, indexed by process types or process identifiers in case of multi-instance support. This is similar to what we did for the communication medium (the "substitution" in this case being limited to a single variable, *mnet*) and also on a recent work on adding space information to BPMN [205]. However, the treatment for unbound data (*e.g.*, if one wants to verify a process, whatever the initialization of the data objects is, or with data stores whose content is unknown) is much more complicated. This could be tackled using approaches based on symbolic verification [173, 211–214].
- **Multi-instance support.** BPMN multi-instance constructs for processes (pool lanes) and activities (subprocesses and tasks). Supporting these constructs requires an extended format for the tokens to carry process identifiers types with possible specific indexed structures and the support of data constructs. This would be reflected in the semantic rules for the BPMN constructs and adds a degree to the complexity of the semantics. Some approaches that may be taken as references to deal with data, multi-instance activities and multi-instance pool lanes are [71, 138].

Moreover, we identified some issues while experimenting timed-BPMN semantics with Alloy implementation:

- **Automated time tool support.** The main problem with dealing with time is fundamental. This applies as well to BPMN. With time, the state space explodes naturally. Alloy cannot handle this without any form of abstraction. Yet, one still has to find a better representation of time steps in it in order to make automated verification amenable.
- **Analysis results.** We have noticed two difficulties using the Alloy Analyser as a verification tool: (1) estimate the length of traces necessary to validate a property; (2) inefficiency (difficulty to exceed twenty states). This second difficulty is the one that makes Alloy unsuitable for checking properties with time, except for small examples. Alloy is better suited for checking structural properties for dynamic ones. Therefore, we plan to study the use of other formalisms like Why3 or SMT which benefit from a symbolic representation and a high level of abstraction or formalisms like Timed Automata and Time(d) Petri nets which benefit from a strong community that focuses on analysing such models and study methods to mitigate the state space explosion explicitly.

Finally, some ideas for the improvement of the fbpmn tool would be addressed:

- **Properties.** The properties whose verification is supported by fbpmn tool-suite are generic ones and do not consider time aspects, but for classic time properties such as execution time (min, max). We expect to extend the support to other time properties such as execution times average, waiting times, or synchronisation times.
- **More experiments.** To verify the BPMN models using the Alloy Analyser, we have chosen the default solver, Minisat, which is suitable used for small problems only. It would be useful to perform the analyses with Berkmin, which seems better for larger problems. On the other hand, during this thesis, we tested the effectiveness of our tool in terms of capacity and execution time. However, we were unable to measure the actual impact of the tool from the perspective of a user in charge of specifying and verifying a business process model. We want to integrate our tool as a plug-in in more general-purpose platforms for business processes, such as Apromore [137] or ProM [66], to expand its use and have feedbacks for improvement.
- **Pattern support.** The objective here is to strengthen the seven timed process pattern integration as a reference in BPMN process modelling tools. We plan to extend on a tool with the formal support of these patterns in the form of a set of rules walk-through that give confidence that they are practically applicable.

Chapter

7 Bibliography

References for Chapter 1: Introduction

[1] Marlon Dumas et al. *Fundamentals of Business Process Management, Second Edition.* Springer, 2018 *Cited on pages 1, 10, 11, 94, 95.*

[2] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures.* Springer, 2007 *Cited on pages 1, 16.*

[3] OMG Group. *Business Process Modeling Notation.* 2013. URL: `http://www.omg.org/spec/BPMN/2.0.2/` *Cited on pages 1, 3, 4, 12, 20, 21, 58, 72, 90, 91, 99, 109.*

[4] W.M.P. van der Aalst. "Formalization and Verification of Event-Driven Process Chains". In: *Information and Software technology* 41 (1999), pp. 639–650. DOI: `10.1016/S0950-5849(99)00016-6` *Cited on pages 1, 12.*

[5] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. "YAWL: Yet Another Workflow Language". In: *Information systems* 30 (2005), pp. 245–275. DOI: `10.1016/j.is.2004.02.002` *Cited on pages 1, 12.*

[6] Marlon Dumas and Arthur H. M. ter Hofstede. "UML Activity Diagrams as a Workflow Specification Language". In: *4th International Conference on the unified Modeling Language.* Ed. by Martin Gogolla and Cris Kobryn. 2001, pp. 76–90. DOI: `10.1007/3-540-45441-1\_7` *Cited on pages 1, 12.*

[7] Manfred Reichert and Barbara Weber. *Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies.* Springer, 2012 *Cited on page 1.*

[8] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures.* Springer Verlag Berlin Heidelberg, 2012 *Cited on pages 1, 10.*

[9] Marcello La Rosa. "Managing variability in process-aware information systems". PhD thesis. Queensland University of Technology, 2009 *Cited on page 2.*

[10] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. "Introduction to Model Checking". In: *Handbook of Model Checking.* 2018, pp. 1–26. DOI: `10.1007/978-3-319-10575-8_1` *Cited on pages 2, 27.*

[11] Basit Mubeen Abdul et al. " UBBA: Unity Based BPMN Animator". In: *Information Systems Engineering in Responsible Information Systems, CAISE Forum.* 2019, pp. 1–9. DOI: `10.1007/978-3-030-21297-1\_1` *Cited on page 2.*

[12] Fabio Casati et al. "Towards Business Processes Orchestrating the Physical Enterprise with Wireless Sensor Networks". In: *34th International Conference on Software Engineering, ICSE.* 2012, pp. 1357–1360. DOI: `10.1109/DCOSS.2011.5982159` *Cited on page 3.*

[13] Sonja Meyer, Andreas Ruppen, and Lorenz M. Hilty. "The Things of the Internet of Things in BPMN". In: *Advanced Information Systems Engineering, CAiSE Workshops.* 2015, pp. 285–297. DOI: `10.1007/978-3-319-19243-7_27` *Cited on page 3.*

[14] Saoussen Cheikhrouhou et al. “Toward a Time-centric modeling of Business Processes in BPMN 2.0”. In: *The 15th International Conference on Information Integration and Web-based Applications & Services, IIWAS*. 2013, pp. 154–163. DOI: 10.1145/2539150.2539182 *Cited on pages 3, 152.*

[15] Andreas Lanz, Barbara Weber, and Manfred Reichert. “Time patterns for process-aware information systems”. In: *Requirements Engineering* 19 (2014), pp. 113–141. DOI: 10.1007/s00766-012-0162-3 *Cited on pages 3, 99, 114, 115, 117, 118, 120, 122, 123.*

[16] *ISO 8601:2004, Data elements and interchange formats — Information interchange — Representation of dates and times*. Standard. ISO, 2004 *Cited on pages 4, 99.*

[17] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison Wesley, 2002 *Cited on pages 4, 27, 28, 127.*

[18] Alcino Cunha. “Bounded Model Checking of Temporal Formulas with Alloy”. In: *4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, ABZ*. 2014, pp. 303–308. DOI: 10.1007/978-3-662-43652-3_29 *Cited on page 4.*

References for Chapter 2: Background

[1] Marlon Dumas et al. *Fundamentals of Business Process Management, Second Edition*. Springer, 2018 *Cited on pages 1, 10, 11, 94, 95.*

[2] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007 *Cited on pages 1, 16.*

[3] OMG Group. *Business Process Modeling Notation*. 2013. URL: http://www.omg.org/spec/BPMN/2.0.2/ *Cited on pages 1, 3, 4, 12, 20, 21, 58, 72, 90, 91, 99, 109.*

[4] W.M.P. van der Aalst. “Formalization and Verification of Event-Driven Process Chains”. In: *Information and Software technology* 41 (1999), pp. 639–650. DOI: 10.1016/S0950-5849(99)00016-6 *Cited on pages 1, 12.*

[5] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. “YAWL: Yet Another Workflow Language”. In: *Information systems* 30 (2005), pp. 245–275. DOI: 10.1016/j.is.2004.02.002 *Cited on pages 1, 12.*

[6] Marlon Dumas and Arthur H. M. ter Hofstede. “UML Activity Diagrams as a Workflow Specification Language”. In: *4th International Conference on the unified Modeling Language*. Ed. by Martin Gogolla and Cris Kobryn. 2001, pp. 76–90. DOI: 10.1007/3-540-45441-1_7 *Cited on pages 1, 12.*

[8] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer Verlag Berlin Heidelberg, 2012 *Cited on pages 1, 10.*

[10] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. “Introduction to Model Checking”. In: *Handbook of Model Checking*. 2018, pp. 1–26. DOI: 10.1007/978-3-319-10575-8_1 *Cited on pages 2, 27.*

[17] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison Wesley, 2002 *Cited on pages 4, 27, 28, 127.*

[19] Chris Newcombe. “Why Amazon Chose TLA +”. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ, 2014, Proceedings*. Vol. 8477. 2014, pp. 25–39. DOI: 10.1007/978-3-662-43652-3_3 *Cited on page 9.*

[20] Hamid Bagheri et al. “Detection of Design Flaws in the Android Permission Protocol Through Bounded Verification”. In: *20th International Symposium Formal Methods, FM*. 2015, pp. 73–89. DOI: 10.1007/978-3-319-19249-9_6 *Cited on page 9.*

[21] Mathias Weske. *Business Process Management - Concepts, Languages, Architectures, Third Edition.* Springer, 2019 *Cited on page 10.*

[22] James Lyle Peterson. *Petri net theory and the modeling of systems.* Prentice Hall PTR, 1981 *Cited on page 12.*

[23] Flavio Corradini et al. "A Classification of BPMN Collaborations based on Safeness and Soundness Notions". In: *Proceedings of the 25th International Workshop on Expressiveness in Concurrency,EPTCS.* 2018, pp. 37–52. DOI: 10.4204/EPTCS.276.5 *Cited on pages 15, 17, 94–96, 132, 138, 150, 151.*

[24] OMG Group. *BPMN 2.0 by Example.* 2010. URL: http://www.iet.unipi.it/m.cimino/gpa/res/BPMN_by_example.pdf *Cited on page 20.*

[25] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, eds. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001].* Vol. 2500. 2002. DOI: 10.1007/3-540-36387-4 *Cited on page 23.*

[26] Christos Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994 *Cited on page 23.*

[27] Hendrik Decker. "Foundations of First-order Databases". In: *Proceedings of the First Compulog Net Meeting on Knowledge Bases, CNKBS'92.* 1992, pp. 6–8. DOI: 10.1145/27629.27630 *Cited on page 23.*

[28] Herbert B. Enderton. *A mathematical Introduction to Logic.* Academic Press, 1972 *Cited on page 23.*

[29] Heinz-Dieter Ebbinghaus, Jörg Flum, and Wolfgang Thomas. *Mathematical Logic (2nd Edition).* Springer, 1994 *Cited on page 23.*

[30] Melvin Fitting. *First-Order Logic and Automated Theorem Proving, Second Edition.* Springer, 1996 *Cited on page 23.*

[31] Dirk van Dalen. *Logic and structure (4th Edition).* Universitext. Springer, 2008 *Cited on page 23.*

[32] Paul Ammann and Jeff Offutt. *Introduction to Software Testing.* Cambridge University Press, 2008 *Cited on page 26.*

[33] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, POPL.* 1977, pp. 238–252. DOI: 10.1145/512950.512973 *Cited on page 27.*

[34] Tobias Nipkow and Leonor Prensa Nieto. "Owicki/Gries in Isabelle/HOL". In: *Second Internationsl Conference Fundamental Approaches to Software Engineering, FASE.* Vol. 1577. 1999, pp. 188–203. DOI: 10.1007/978-3-540-49020-3_13 *Cited on page 27.*

[35] William Landi. "Undecidability of Static Analysis". In: *LOPLAS* 1 (1992), pp. 323–337. DOI: 10.1145/161494.161501 *Cited on page 27.*

[36] Edsger W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In: *Communications of the ACM* 18 (1975), pp. 453–457. DOI: 10.1145/360933.360975 *Cited on page 27.*

[37] Lawrence C. Paulson. *Isabelle - A Generic Theorem Prover.* Springer, 1994 *Cited on page 27.*

[38] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic.* Springer, 2002 *Cited on page 27.*

[39] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Communication ACM* 12 (1969), pp. 576–580. DOI: 10.1145/1562764.1562779 *Cited on pages 27, 80.*

[40] Edmund M. Clarke et al. “Model Checking and the State Explosion Problem”. In: *Tools for Practical Software Verification, LASER, International Summer School 2011, Revised Tutorial Lectures.* 2011, pp. 1–30. DOI: 10.1007/978-3-642-35746-6_1 *Cited on page 27.*

[41] Gerard Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004 *Cited on page 27.*

[42] Sylvain Conchon et al. “Cubicle: A Parallel SMT-Based Model Checker for Parameterized Systems - Tool Paper”. In: *24th International Conference Computer Aided Verification, CAV.* 2012, pp. 718–724. DOI: 10.1007/978-3-642-31424-7_55 *Cited on page 27.*

[43] Johan Bengtsson et al. “UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems”. In: *Hybrid Systems III: Verification and Control.* 1995, pp. 232–243. DOI: 10.1007/BFb0020949 *Cited on page 27.*

[44] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. “Model Checking TLA$^+$ Specifications”. In: *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME.* Vol. 1703. 1999, pp. 54–66. DOI: 10.1007/3-540-48153-2_6 *Cited on page 33.*

[45] Leslie Lamport. *The TLA Home Page.* 2018. URL: http://lamport.azurewebsites.net/tla/tla.html *Cited on page 33.*

[46] Mana Taghdiri and Daniel Jackson. “A Lightweight Formal Analysis of a Multicast Key Management Scheme”. In: *Formal Techniques for Networked and Distributed Systems - FORTE 2003, 23rd IFIP WG 6.1 International Conference, 2003, Proceedings.* 2003, pp. 240–256. DOI: 10.1007/978-3-540-39979-7_16 *Cited on pages 34, 127.*

[47] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, 2012 *Cited on pages 34, 36, 38, 132.*

[48] Marcelo F. Frias et al. “Reasoning About Static and Dynamic Properties in Alloy: A Purely Relational Approach”. In: *ACM Transition Software Engineering Methodology* 14 (2005), pp. 478–526. DOI: 10.1145/1101815.1101819 *Cited on pages 34, 36.*

[49] Daniel Jackson. “Alloy: A Lightweight Object Modelling Notation”. In: *ACM Transactions on Software Engineering and Methodology* 11 (2002), pp. 256–290. DOI: 10.1145/505145.505149 *Cited on page 34.*

[50] Emina Torlak and Daniel Jackson. “Kodkod: A Relational Model Finder”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS, 2007, Proceedings.* Vol. 4424. 2007, pp. 632–647. DOI: 10.1007/978-3-540-71209-1_49 *Cited on page 38.*

[51] Daniel Le Berre and Stéphanie Roussel. “Sat4j 2.3.2: on the Fly solver Configuration System Description”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 8 (2014), pp. 197–202. DOI: 10.3233/sat190098 *Cited on page 38.*

[52] Niklas Eén and Niklas Sörensson. “An Extensible SAT-solver”. In: *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT , 2003.* Vol. 2919. 2003, pp. 502–518 *Cited on page 38.*

[53] Eugene Goldberg and Yakov Novikov. “BerkMin: A Fast and Robust Sat-solver”. In: *Discrete Applied Mathematics* 155 (2007), pp. 1549–1561. DOI: 10.1016/j.dam.2006.10.007 *Cited on page 38.*

[54] Nuno Macedo and Alcino Cunha. “Alloy meets TLA+: An Exploratory Study”. In: *CoRR* abs/1603.03599 (2016) *Cited on page 39.*

References for Chapter 3: Literature Review

[3] OMG Group. *Business Process Modeling Notation.* 2013. URL: http://www.omg.org/spec/BPMN/2.0.2/ *Cited on pages 1, 3, 4, 12, 20, 21, 58, 72, 90, 91, 99, 109.*

[55] Shoichi Morimoto. “A Survey of Formal Verification for Business Process Modeling”. In: *8th International Conference on Computational Science, ICCS.* 2008, pp. 514–522. DOI: 10.1007/978-3-540-69387-1_58 *Cited on pages 41–43.*

[56] Wil MP Van der Aalst. “Business Process Management: a Comprehensive Survey”. In: *ISRN Software Engineering* 2013 (2013). DOI: 10.1155/2013/507984 *Cited on page 41.*

[57] Michael Fellmann and Andrea Zasada. “State-of-the-art of Business Process Compliance Approaches: A Survey”. In: *EMISA Forum* 36 (2016), pp. 45–48 *Cited on pages 41–43.*

[58] Hanh H Hoang, Jason J Jung, and Chi P Tran. “Ontology-based Approaches for Cross-Enterprise Collaboration: a Literature Review on Semantic Business Process Management”. In: *Enterprise Information Systems* 8 (2014), pp. 648–664. DOI: 10.1080/17517575.2013.767382 *Cited on page 41.*

[59] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. “Guidelines for Conducting Systematic Mapping Studies in Software Engineering: An Update”. In: *Information and Software Technology* 64 (2015), pp. 1–18. DOI: 10.1016/j.infsof.2015.03.007 *Cited on page 42.*

[60] Volker Gruhn and Ralf Laue. “A Comparison of Soundness Results Obtained by Different Approaches”. In: *International Business Process Management Workshops, BPM.* 2009, pp. 501–512. DOI: 10.1007/978-3-642-12186-9_47 *Cited on page 43.*

[61] Anam Amjad et al. “Event-Driven Process Chain for Modeling and Verification of Business Requirements-A Systematic Literature Review”. In: *IEEE Access* 6 (2018), pp. 9027–9048. DOI: 10.1109/ACCESS.2018.2791666 *Cited on page 43.*

[62] Heerko Groefsema and Doina Bucur. “A Survey of Formal Business Process Verification: From Soundness to Variability”. In: *3rd International Symposium on Business Modeling and Software Design, BMSD.* 2013, pp. 198–203. DOI: 10.5220/0004775401980203 *Cited on page 43.*

[63] Abiodun Muyideen Mustapha et al. “A Systematic Literature Review on Compliance Requirements Management of Business Processes”. In: *International Journal of System Assurance Engineering and Management* 11 (2020), pp. 561–576. DOI: 10.1007/s13198-020-00985-w *Cited on page 43.*

[64] Remco M Dijkman, Marlon Dumas, and Chun Ouyang. “Semantics and Analysis of Business Process Models in BPMN”. In: *Information and Software technology* 50 (2008), pp. 1281–1294. DOI: 10.1016/j.infsof.2008.02.006 *Cited on pages 46, 47, 54–57, 59, 60, 62, 150, 151.*

[65] Remco Dijkman. *Transformer.* Not Found. 2008. URL: https://is.tm.tue.nl/staff/rdijkman/cbd.html:transformer *Cited on pages 46, 59.*

[66] Eindhoven University of Technology Process Mining Group. *Process Mining Framework (PROM).* 2020. URL: http://www.processmining.org/prom/start *Cited on pages 46, 153.*

[67] Tsukasa Takemura. “Formal Semantics and Verification of BPMN Transaction and Compensation”. In: *Proceedings of the 3rd IEEE Asia-Pacific Services Computing Conference, APSCC.* 2008, pp. 284–290. DOI: 10.1109/APSCC.2008.208 *Cited on pages 46, 54, 56, 57.*

[68] C Ou-Yang and YD Lin. "BPMN-based Business Process Model Feasibility Analysis: a Petri net Approach". In: *International Journal of Production Research* 46 (2008), pp. 3763–3781. DOI: 10.1080/00207540701199677 *Cited on pages 46, 54, 55, 57.*

[69] Christian Wolter, Philip Miseldine, and Christoph Meinel. "Verification of Business Process Entailment Constraints Using SPIN". In: *Engineering Secure Software and Systems, First International Symposium ESSoS*. 2009, pp. 1–15. DOI: 10.1007/978-3-642-00199-4_1 *Cited on pages 46, 54, 55, 57, 59, 60, 62.*

[70] Remco M. Dijkman and Pieter Van Gorp. "BPMN 2.0 Execution Semantics Formalized as Graph Rewrite Rules". In: *Proceedings of Second International Workshop of Business Process Modeling Notation, BPMN*. 2010, pp. 16–30. DOI: 10.1007/978-3-642-16298-5_4 *Cited on pages 46, 54, 55, 57.*

[71] Pieter Van Gorp and Remco M. Dijkman. "A Visual Token-based Formalization of BPMN 2.0 based on In-place Transformations". In: *Information and Software Technology* 55 (2013), pp. 365–394. DOI: 10.1016/j.infsof.2012.08.014 *Cited on pages 46, 54, 55, 57–60, 62, 150, 151, 153.*

[72] Alberto Bastias, Sidharth Bihary, and Suman Roy. "An Automated Analysis of Errors for BPM Processes Modeled using an In-House Infosys Tool". In: *18th Asia Pacific Software Engineering Conference, APSEC*. 2011, pp. 97–105. DOI: 10.1109/APSEC.2011.49 *Cited on pages 46, 54, 55, 57.*

[73] The Woflan Development Team. *The Petri-net-based workflow analyzer*. 2004. URL: http://www.swmath.org/software/7028 *Cited on page 46.*

[74] Ben D'Angelo et al. "LOLA: Runtime Monitoring of Synchronous Systems". In: *12th International Symposium on Temporal Representation and Reasoning, TIME*. 2005, pp. 166–174. DOI: 10.1109/TIME.2005.26 *Cited on page 46.*

[75] Suman Roy et al. "An Empirical Study of Error Patterns in Industrial Business Process Models". In: *IEEETransactions on Services Computing* (2013), pp. 140–153. DOI: 10.1109/TSC.2013.10 *Cited on pages 46, 54, 55, 57.*

[76] Suman Roy, A. S. M. Sajeev, and Srivibha Sripathy. "Diagnosing Industrial Business Processes: Early Experiences". In: *19th International Symposium Formal Methods Proceedings, FM*. 2014, pp. 703–717. DOI: 10.1007/978-3-319-06410-9_47 *Cited on pages 46, 54, 55, 57.*

[77] Suman Roy and A. S. M. Sajeev. "A Formal Framework for Diagnostic Analysis for Errors of Business Processes". In: *Transactions on Petri Nets and Other Models of Concurrency* 11 (2016), pp. 226–261. DOI: 10.1007/978-3-662-53401-4_11 *Cited on pages 47, 54, 55, 57.*

[78] Jutta A. Mülle, Christine Tex, and Klemens Böhm. "A Practical Data-flow Verification Scheme for Business Processes". In: *Information Systems* 81 (2019), pp. 136–151. DOI: 10.1016/j.is.2018.12.002 *Cited on pages 47, 54, 55, 57.*

[79] Geneva ISO. *Switzerland: Road vehicles–Open Test sequence eXchange format (OTX)*. 2012. URL: https://www.iso.org/standard/53509.html *Cited on page 47.*

[80] Elaheh Ordoni, Jutta A. Mülle, and Klemens Böhm. "Verification of Data-Value-Aware Processes and a Case Study on Spectrum Auctions". In: *22nd IEEE Conference on Business Informatics, CBI*. 2020, pp. 181–190. DOI: 10.1109/CBI49978.2020.00027 *Cited on pages 47, 54, 55, 57.*

[81] Carlo Combi, Barbara Oliboni, and Francesca Zerbato. "A Modular Approach to the Specification and Management of Time Duration Constraints in BPMN". In: *Information Systems* 84 (2019), pp. 111–144. DOI: 10.1016/j.is.2019.04.010 *Cited on pages 47, 54, 57, 94, 150, 152.*

[82] Anass Rachdi, Abdeslam En-Nouaary, and Mohamed Dahchour. “Liveness and Reachability Analysis of BPMN Process Models”. In: *Journal of Computing and Information Technology* 24 (2016), pp. 195–207. DOI: 10.20532/CIT.2016.1002774 *Cited on pages 47, 54, 55, 57.*

[83] Said Meghzili et al. “Transformation and Validation of BPMN Models to Petri Nets Models Using GROOVE”. In: *International Conference on Advanced Aspects of Software Engineering, ICAASE.* 2016, pp. 22–29. DOI: 10.1109/ICAASE.2016.7843859 *Cited on pages 47, 54, 55, 57.*

[84] Said Meghzili et al. “An Approach for the Transformation and Verification of BPMN Models to Colored Petri Nets Models”. In: *International Journal of Software Innovation* 8 (2020), pp. 17–49. DOI: 10.4018/IJSI.2020010102 *Cited on pages 47, 54, 55, 57.*

[85] Ahmed Kheldoun, Kamel Barkaoui, and Malika Ioualalen. “Specification and Verification of Complex Business Processes – a High-Level Petri Net-Based Approach”. In: *13th International Conference on Business Process Management, BPM.* 2016, pp. 55–71. DOI: 10.1007/978-3-319-23063-4_4 *Cited on pages 47, 54, 57.*

[86] Ahmed Kheldoun, Kamel Barkaoui, and Malika Ioualalen. “Formal Verification of Complex Business Processes Based on High-level Petri Nets”. In: *Information Sciences* 385 (2017), pp. 39–54. DOI: 10.1016/j.ins.2016.12.044 *Cited on pages 47, 54, 57, 59, 60, 62, 150, 151.*

[87] Umair Mutarraf et al. “Transformation of Business Process Model and Notation Models onto Petri Nets and their Analysis”. In: *Advances in Mechanical Engineering* 10 (2018), p. 1687814018808170. DOI: 10.1177/1687814018808170 *Cited on pages 47, 54, 55, 57.*

[88] Reggie Davidrajuh. *GPenSIM: a general purpose Petri net simulator.* 2015. URL: http://www.davidrajuh.net/gpensim *Cited on page 48.*

[89] Chanon Dechsupa, Wiwat Vatanawood, and Arthit Thongtak. “Transformation of the BPMN Design Model into a Colored Petri Net using the Partitioning Approach”. In: *IEEE Access* 6 (2018), pp. 38421–38436. DOI: 10.1109/ACCESS.2018.2853669 *Cited on pages 48, 54, 57, 59, 60, 62, 150, 151.*

[90] Chanon Dechsupa, Wiwat Vatanawood, and Arthit Thongtak. “Hierarchical Verification for the BPMN Design Model Using State Space Analysis”. In: *IEEE Access* 7 (2019), pp. 16795–16815 *Cited on pages 48, 54.*

[91] Saoussen Cheikhrouhou et al. “Formal Specification and Verification of Cloud Resource Allocation using Timed Petri-Nets”. In: *International Workshops on New Trends in Model and Data Engineering Proceedings, MEDI.* 2018, pp. 40–49. DOI: 10.1007/978-3-030-02852-7_4 *Cited on pages 48, 54, 57.*

[92] Stephan Haarmann and Mathias Weske. “Cross-Case Data Objects in Business Processes: Semantics and Analysis”. In: *Business Process Management Forum, BPM Forum.* 2020, pp. 3–17. DOI: 10.1007/978-3-030-58638-6_1 *Cited on pages 48, 54, 57, 59, 60, 62.*

[93] Stephan Haarmann and Mathias Weske. *fcm2cpn.* 2020. URL: https://bptlab.github.io/fcm2cpn/ *Cited on pages 48, 59.*

[94] JianHong Ye et al. “Transformation of BPMN to YAWL”. In: *International Conference on Computer Science and Software Engineering, CSSE.* 2008, pp. 354–359. DOI: 10.1109/APSCC.2008.208 *Cited on pages 48, 54, 57, 59, 60, 62.*

[95] JianHong Ye et al. “Formal Semantics of BPMN Process Models using YAWL”. In: *Second International Symposium on Intelligent Information Technology Application.* 2008, pp. 70–74. DOI: 10.1109/IITA.2008.68 *Cited on pages 48, 54, 57, 59, 60, 62, 150, 151.*

[96] JianHong Ye and Wen Song. "Transformation of BPMN Diagrams to YAWL Nets". In: *Journal of Software* 5 (2010), pp. 396–404. DOI: 10.4304/jsw.5.4.396-404 *Cited on pages 48, 54, 57, 59, 60, 62.*

[97] Oussama Mohammed Kherbouche, Adeel Ahmad, and Henri Basson. "Using Model Checking to Control the Structural Errors in BPMN Models". In: *IEEE 7th International Conference on Research Challenges in Information Science, RCIS*. 2013, pp. 1–12. DOI: 10.1109/RCIS.2013.6577723 *Cited on pages 48, 54–57, 59, 60, 62.*

[98] Oussama Mohammed Kherbouche, Ahmad Adeel, and Henri Basson. "Detecting Structural Errors in BPMN Process Models". In: *15th IEEE International Multitopic Conference, INMIC*. 2012, pp. 425–431. DOI: 10.1109/INMIC.2012.6511490 *Cited on pages 48, 54, 55, 57, 59, 60, 62.*

[99] Kenji Watahiki, Fuyuki Ishikawa, and Kunihiko Hiraishi. "Formal Verification of Business Processes with Temporal and Resource Constraints". In: *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, SMC*. 2011, pp. 1173–1180. DOI: 10.1109/ICSMC.2011.6083857 *Cited on pages 49, 54, 55, 57, 150, 152.*

[100] Kelly Rosa Braghetto, João Eduardo Ferreira, and Jean-Marc Vincent. "Performance Evaluation of Business Processes through a Formal Transformation to SAN". In: *8th European Performance Engineering Workshop, EPEW*. 2011, pp. 42–56. DOI: 10.1007/978-3-642-24749-1_5 *Cited on pages 49, 54, 55, 57, 59, 60, 62.*

[101] Luis E. Mendoza Morales. "Business Process Verification Using a Formal Compositional Approach and Timed Automata". In: *XXXIX Latin American Computing Conference, CLEI*. 2013, pp. 1–10. DOI: 10.1109/CLEI.2013.6670616 *Cited on pages 49, 54, 55, 57, 59, 60, 62.*

[102] Luis Mendoza. "Business Process Verification: The Application of Model Checking and Timed Automata". In: *CLEI Electronic Journal* 17 (2014). DOI: 10.19153/cleiej.17.2.2 *Cited on pages 49, 54–57, 59, 60, 62.*

[103] Aleksander González et al. " BTRANSFORMER - A Tool for BPMN to CSP+T Transformation". In: *Proceedings of the 13th International Conference on Enterprise Information Systems, ICEIS*. 2011, pp. 363–366. DOI: 10.5220/0003430003630366 *Cited on page 49.*

[104] Saoussen Cheikhrouhou et al. "Enhancing Formal Specification and Verification of Temporal Constraints in Business Processes". In: *IEEE International Conference on Services Computing, SCC*. 2014, pp. 701–708. DOI: 10.1109/SCC.2014.97 *Cited on pages 49, 54, 55, 57, 150, 152.*

[105] Ralph Hoch et al. "Verification of Business Processes Against Business Rules Using Object Life Cycles". In: *New Advances in Information Systems and Technologies - Volume 1, WorldCIST*. 2016, pp. 589–598. DOI: 10.1007/978-3-319-31232-3_55 *Cited on pages 49, 54, 55, 57.*

[106] Sihem Mallek et al. "Enabling Model Checking for Collaborative Process Analysis: from BPMN to 'Network of Timed Automata". In: *Enterprise Information Systems* 9 (2015), pp. 279–299. DOI: 10.1080/17517575.2013.879211 *Cited on pages 49, 54–57.*

[107] Pascal Poizat, Gwen Salaün, and Ajay Krishna. "Checking Business Process Evolution". In: *13th International Conference on Formal Aspects of Component Software,FACS*. 2016, pp. 36–53. DOI: 10.1007/978-3-319-57666-4_4 *Cited on pages 49, 54, 55.*

[108] Ajay Krishna, Pascal Poizat, and Gwen Salaün. "Checking Business Process Evolution". In: *Science of Computer Programming* 170 (2019), pp. 1–26. DOI: 10.1016/j.scico.2018.09.007 *Cited on pages 49, 54, 138.*

[109] Ajay Krishna, Pascal Poizat, and Gwen Salaün. “VBPMN: Automated Verification of BPMN Processes (Tool Paper)”. In: *13th International Conference on integrated Formal Methods, iFM*. 2017, pp. 323–331. DOI: 10.1007/978-3-319-66845-1_21 *Cited on pages 49, 54, 55, 57, 59, 60, 63.*

[110] Salma Ayari, Yousra Bendaly Hlaoui, and Leila Jemni Ben Ayed. “A Refinement based Verification Approach of BPMN Models Using NuSMV”. In: *Proceedings of the 13th International Conference on Software Technologies, ICSOFT*. 2018, pp. 563–574. DOI: 10.5220/0006914105630574 *Cited on pages 50, 54, 55, 57.*

[111] Mihal Brumbulli, Emmanuel Gaudin, and Ciprian Teodorov. “Automatic Verification of BPMN Models”. In: *10th European Congress on Embedded Real Time Software and Systems, ERTS*. 2020, pp. 1–6 *Cited on pages 50, 54, 55, 57, 59, 60, 62.*

[112] VeriMoB project team. *BPMN Formalization*. 2019. URL: http://www.obpcdl.org *Cited on pages 50, 59.*

[113] Marco Autili, Paola Inverardi, and Patrizio Pelliccione. “Graphical Scenarios for Specifying Temporal Properties: an Automated Approach”. In: *Automated Software Engineering* 14 (2007), pp. 293–340. DOI: 10.1007/s10515-007-0012-6 *Cited on page 50.*

[114] Peter Y. H. Wong and Jeremy Gibbons. “A Process Semantics for BPMN”. In: *Formal Methods and Software Engineering, 10th International Conference on Formal Engineering Methods, ICFEM*. 2008, pp. 355–374. DOI: 10.1007/978-3-540-88194-0_22 *Cited on pages 50, 54–57, 59, 60, 62.*

[115] Peter YH Wong and Jeremy Gibbons. “Verifying Business Process Compatibility (short paper)”. In: *Proceedings of the Eighth International Conference on Quality Software, QSIC*. 2008, pp. 126–131. DOI: 10.1109/QSIC.2008.6 *Cited on pages 50, 54, 55, 57.*

[116] Peter YH Wong and Jeremy Gibbons. “A Relative Timed Semantics for BPMN”. In: *Electronic Notes in Theoretical Computer Science* 229 (2009), pp. 59–75. DOI: 10.1016/j.entcs.2009.06.029 *Cited on pages 50, 54, 55, 57, 150, 152.*

[117] Peter Wong YH and Jeremy Gibbons. “Formalisations and Applications of BPMN”. In: *Science of Computer Programming* 76 (2011), pp. 633–650. DOI: 10.1016/j.scico.2009.09.010 *Cited on pages 50, 54, 55, 57, 150, 151.*

[118] Davide Prandi, Paola Quaglia, and Nicola Zannone. “Formal Analysis of BPMN via a Translation into COWS”. In: *Proceedings of 10th International Conference Coordination Models and Languages, COORDINATION*. 2008, pp. 249–263. DOI: 10.1007/978-3-540-68265-3_16 *Cited on pages 50, 54, 55, 57.*

[119] Manuel I. Capel and Luis Eduardo Mendoza. “Automating the Transformation from BPMN Models to CSP+ T Specifications”. In: *35th Annual IEEE Software Engineering Workshop, SEW*. 2012, pp. 100–109. DOI: 10.1109/SEW.2012.17 *Cited on pages 50, 54, 55, 57, 59, 60, 63, 150, 152.*

[120] Luke Herbert and Robin Sharp. “Using Stochastic Model Checking to Provision Complex Business Services”. In: *14th International IEEE Symposium on High-Assurance Systems Engineering, HASE*. 2012, pp. 98–105. DOI: 10.1109/HASE.2012.29 *Cited on pages 50, 54, 57, 59, 60, 62.*

[121] Luke T Herbert, Zaza Hansen, and Peter Jacobsen. “SBAT: A Stochastic BPMN Analysis Tool”. In: *Proceedings of the ASME 2014 12th Biennial Conference on Engineering Systems Design and Analysis, ESDA*. 2014, p. 10. DOI: 10.1115/ESDA2014-20437 *Cited on pages 50, 54, 55, 57, 59, 60, 62.*

[122] Luke Thomas Herbert and Zaza Nadja Lee Hansen. "Restructuring of Workflows to Minimise Errors via Stochastic Model Checking: An Automated Evolutionary Approach". In: *Reliability Engineering & System Safety* 145 (2016), pp. 351–365. DOI: `10.1016/j.ress.2015.07.002` *Cited on pages 50, 54, 55, 57.*

[123] Sofiane Boukelkoul, Ramdane Maamri, and Mohammed Chihoub. "A Discrete Event Model for Analysis and Verification of Time-Constrained Business Processes". In: *Concurrency and Computation: Practice and Experience* 33 (2021). DOI: `10.1002/cpe.5753` *Cited on pages 51, 54, 55, 57.*

[124] Vitus S. W. Lam. "Formal Analysis of BPMN Models: a NuSMV-Based Approach". In: *International Journal of Software Engineering and Knowledge Engineering* 20 (2010), pp. 987–1023. DOI: `10.1142/S0218194010005079` *Cited on pages 51, 54, 55, 57.*

[125] Vitus SW Lam. "A Precise Execution Semantics for BPMN". In: *International Journal of Computer Science* 39 (2012), pp. 20–33. DOI: `10.1109/MIC.2004.58` *Cited on pages 51, 54, 57, 150, 151.*

[126] Pieter M. Kwantes et al. "Towards Compliance Verification between Global and Local Process Models". In: *8th International Conference on Graph Transformation, ICGT*. 2015, pp. 221–236. DOI: `10.1007/978-3-319-21145-9\_14` *Cited on pages 51, 54, 55.*

[127] Marco Brambilla et al. "The Role of Visual Tools in a Web Application Design and Verification Framework: A Visual Notation for LTL Formulae". In: *Web Engineering, 5th International Conference, ICWE*. 2005, pp. 557–568. DOI: `10.1007/11531371\_70` *Cited on page 51.*

[128] Nissreen A. S. El-Saber and Artur Boronat. "BPMN Formalization and Verification using Maude". In: *Proceedings of the 2014 Workshop on Behaviour Modelling - Foundations and Applications, BM-FA*. 2014, pp. 1–12. DOI: `10.1145/2630768.2630769` *Cited on pages 51, 54, 55, 57, 150, 151.*

[129] Kazuhiro Ogata, Thapana Chaimanont, and Min Zhang. "Formal Modeling and Analysis of Time- and Resource-Sensitive Simple Business Processes". In: *Journal of Information Security and Applications* 31 (2016), pp. 23–40. DOI: `10.1016/j.jisa.2016.05.001` *Cited on pages 51, 54, 55, 57.*

[130] Flavio Corradini et al. "BProVe: a Formal Verification Framework for Business Process Models". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE*. 2017, pp. 217–228. DOI: `10.1109/ASE.2017.8115635` *Cited on pages 51, 54, 55, 57.*

[131] Flavio Corradini et al. "BProVe: Tool Support for Business Process Verification". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE*. 2017, pp. 217–228. DOI: `10.1109/ASE.2017.8115708` *Cited on pages 51, 54, 55, 57, 59, 60, 63.*

[132] Flavio Corradini et al. "A Formal Approach to Modelling and Verification of Business Process Collaborations". In: *Science of Computer Programming* 166 (2018), pp. 35–70. DOI: `10.1016/j.scico.2018.05.008` *Cited on pages 51, 54, 55, 57, 59, 60, 63, 150, 151.*

[133] Flavio Corradini et al. "An Operational Semantics of BPMN Collaboration". In: *12th International Conference Formal Aspects of Component Software, FACS*. 2015, pp. 161–180. DOI: `10.1007/978-3-319-28934-2\_9` *Cited on pages 51, 54, 59, 60, 63.*

[134] Gordon D. Plotkin. "A Structural Approach to Operational Semantics". In: *Journal of Logical and Algebraic Methods in Programming* 60-61 (2004), pp. 17–139. DOI: `10.1016/j.jlap.2004.05.001` *Cited on page 51.*

[135] Flavio Corradini et al. "Global vs. Local Semantics of BPMN 2.0 OR-Join". In: *44th International Conference on Current Trends in Theory and Practice of Computer Science Proceedings, SOFSEM*. 2018, pp. 321–336. DOI: 10.1007/978-3-319-73117-9_23 *Cited on pages 52, 54.*

[136] Fabrizio Fornari et al. "Checking Business Process Correctness in Apromore". In: *Information Systems in the Big Data Era Proceedings, CAiSE Forum*. 2018, pp. 114–123. DOI: 10.1007/978-3-319-92901-9_11 *Cited on pages 52, 54.*

[137] Apromore Pty Ltd. *Apromore*. 2020. URL: https://apromore.org *Cited on pages 52, 153.*

[138] Flavio Corradini et al. "Animating Multiple Instances in BPMN Collaborations: From Formal Semantics to Tool Support". In: *16th International Conference on Business Process Management, BPM*. 2018, pp. 83–101. DOI: 10.1007/978-3-319-98648-7_6 *Cited on pages 52, 54, 59, 60, 63, 138, 150, 151, 153.*

[139] Flavio Corradini et al. "Correctness Checking for BPMN Collaborations with Sub-Processes". In: *Journal of Systems and Software* 166 (2020), p. 110594. DOI: 10.1016/j.jss.2020.110594 *Cited on pages 52, 54–57, 59, 60, 63.*

[140] Flavio Corradini et al. "Well-structuredness, Safeness and Soundness: A Formal Classification of BPMN Collaborations". In: *Journal of Logical and Algebraic Methods in Programming* 119 (2021), p. 100630. DOI: 10.1016/j.jlamp.2020.100630 *Cited on pages 52, 54, 61, 128.*

[141] Flavio Corradini et al. "Collaboration vs. Choreography Conformance in BPMN 2.0: From Theory to Practice". In: *22nd IEEE International Enterprise Distributed Object Computing Conference, EDOC*. 2018, pp. 95–104. DOI: 10.1109/EDOC.2018.00022 *Cited on pages 52, 54, 59, 60, 63.*

[142] Francisco Durán and Gwen Salaün. "Verifying Timed BPMN Processes Using Maude". In: *19th IFIP International Conference on Coordination Models and Languages, , COORDINATION*. 2017, pp. 219–236. DOI: 10.1007/978-3-319-59746-1_12 *Cited on pages 52, 54, 55, 57, 59–61, 63, 150, 152.*

[143] Francisco Durán, Camilo Rocha, and Gwen Salaün. "Stochastic analysis of BPMN with time in rewriting logic". In: *Science of Computer Programming* 168 (2018), pp. 1–17. DOI: 10.1016/j.scico.2018.08.007 *Cited on pages 52, 54, 55, 57, 59–61, 63, 150, 152.*

[144] Francisco Durán, Camilo Rocha, and Gwen Salaün. "Computing the Parallelism Degree of Timed BPMN Processes". In: *Collocated Workshops of Software Technologies: Applications and Foundations - STAF*. 2018, pp. 320–335. DOI: 10.1007/978-3-030-04771-9_24 *Cited on pages 52, 54, 57, 59–61, 63.*

[145] Francisco Durán, Camilo Rocha, and Gwen Salaün. "A Rewriting Logic Approach to Resource Allocation Analysis in Business Process Models". In: *Science of Computer Programming* 183 (2019), p. 102303. DOI: 10.1016/j.scico.2019.102303 *Cited on pages 52, 54, 57, 59–61, 63.*

[146] Francisco Durán, Camilo Rocha, and Gwen Salaün. "Symbolic Specification and Verification of Data-Aware BPMN Processes Using Rewriting Modulo SMT". In: *12th International Workshop of Rewriting Logic and Its Applications, WRLA*. 2018, pp. 76–97. DOI: 10.1007/978-3-319-99840-4_5 *Cited on pages 52, 54, 57, 61.*

[147] Antoni Ligeza and Tomasz Potempa. "AI Approach to Formal Analysis of BPMN Models: Towards a Logical Model for BPMN Diagrams". In: *3rd International Workshop on Advances in Business ICT, ABICT*. 2012, pp. 69–88. DOI: 10.1007/978-3-319-03677-9_5 *Cited on pages 52, 54, 55, 57.*

[148] Emanuele De Angelis et al. “Verification of Time-Aware Business Processes Using Constrained Horn Clauses”. In: *26th International Symposium on Logic-Based Program Synthesis and Transformation , LOPSTR*. 2016, pp. 38–55. DOI: `10.1007/978-3-319-63139-4\_3` *Cited on pages 53–55, 57.*

[149] Emanuele De Angelis et al. “Semantics and Controllability of Time-Aware Business Processes”. In: *Fundamenta Informaticae* 165 (2019), pp. 205–244. DOI: `10.3233/FI-2019-1783` *Cited on pages 53–55, 57, 59, 60, 63.*

[150] Marcin Szpyrka, Grzegorz J. Nalepa, and Krzysztof Kluza. “From Process Models to Concurrent Systems in Alvis Language”. In: *Informatica* 28 (2017), pp. 525–545. DOI: `10.15388/INFORMATICA.2017.143` *Cited on pages 53–55, 57.*

[151] Jeremy W. Bryans and Wei Wei. “Formal Analysis of BPMN Models Using Event-B”. In: *15th International Workshop Formal Methods for Industrial Critical Systems, FMICS*. 2010, pp. 33–49. DOI: `10.1007/978-3-642-15898-8\_3` *Cited on pages 53–55, 57.*

[152] Ahlem Ben Younes et al. “From BPMN2 to Event B: A Specification and Verification Approach of Workflow Applications”. In: *43rd IEEE Annual Computer Software and Applications Conference, COMPSAC*. 2019, pp. 561–566. DOI: `10.1109/COMPSAC.2019.10266` *Cited on pages 53–55, 57, 59, 60, 63.*

[153] Junaid Haseeb et al. “Application of Formal Methods to Modelling and Analysis Aspects of Business Process Reengineering”. In: *Business Process Management Journal* 26 (2020), pp. 548–569. DOI: `10.1108/BPMJ-02-2019-0078` *Cited on pages 53–55, 57.*

[154] Diego Calvanese et al. “Formal Modeling and SMT-Based Parameterized Verification of Data-Aware BPMN”. In: *17th International Conference Business Process Management Proceedings, BPM*. 2019, pp. 157–175. DOI: `10.1007/978-3-030-26619-6_12` *Cited on pages 53–55, 57.*

[155] Damiano Falcioni et al. “Direct Verification of BPMN Processes Through an Optimized Unfolding Technique”. In: *12th International Conference on Quality Software, ICQS*. 2012, pp. 179–188. DOI: `10.1109/QSIC.2012.59` *Cited on pages 53, 54, 57, 59, 60, 63.*

[156] Mayssa Bessifi, Ahlem Ben Younes, and Leila Ben Ayed. “BPMN2EVENTB supporting transformation from BPMN 2.0 to Event B using Kermeta”. In: (2021) *Cited on pages 57, 59, 60, 63.*

[157] Business Process Technology research group at the Hasso Plattner Institute. *Oryx*. not found. 2009. URL: `http://oryx-editor.org/` *Cited on page 59.*

[158] Oussama Mohammed Kherbouche, Adeel Ahmad, and Henri Basson. *EPSPIN*. Not Found. 2012. URL: `http://epispin.ewi.tudelft.nl/` *Cited on page 59.*

[159] GrGen.NET developers. *GrGen.NET*. 2010. URL: `http://www.info.uni-karlsruhe.de/software/grgen/` *Cited on page 59.*

[160] Ahmed Kheldoun, Kamel Barkaoui, and Malika Ioualalen. *BPMN Checker*. Not Found. 2016. URL: `--` *Cited on page 59.*

[161] The YAWL Foundation. *BPMN2YAWL*. 2004. URL: `https://yawlfoundation.github.io/` *Cited on page 59.*

[162] Kelly Rosa Braghetto. *BP2SAN*. 2012. URL: `https://www.ime.usp.br/~kellyrb/bp2san/` *Cited on page 59.*

[163] Wong and all. *machine-readable CSP*. 2008. URL: `http://www.cs.ox.ac.uk/peter.wong/bpmn/index.html` *Cited on page 59.*

[164] Barbara et all. *cowslip*. 2012. URL: `https://sourceforge.net/projects/cowslip/` *Cited on page 59.*

[165] PRos project team. *Bprove*. 2017. URL: http://pros.unicam.it/bprove/ *Cited on pages 59, 61.*

[166] Flavio Corradini et al. *MIDA*. 2018. URL: http://pros.unicam.it/mida/ *Cited on page 59.*

[167] PRos project team. *C4*. 2018. URL: http://pros.unicam.it/c4/ *Cited on page 59.*

[168] PROSLabTeam. *S3*. 2018. URL: http://pros.unicam.it/s3/ *Cited on page 59.*

[169] Ajay Krishna, Pascal Poizat, and Gwen Salaün. *VBPMN*. 2016. URL: https://pascalpoizat.github.io/vbpmn-web/ *Cited on page 59.*

[170] Francisco Durán and Gwen Salaün. *Maude Specification and Verification of BPMN Processes*. 2017. URL: http://maude.lcc.uma.es/MaudeBPMN/ *Cited on page 59.*

[171] Francisco Durán and Gwen Salaün. *BPMN Specification*. 2018. URL: http://maude.lcc.uma.es/BPMN-P/ *Cited on page 59.*

[172] Francisco Durán and Gwen Salaün. *Resource Allocation Analysis of BPMN Processes*. 2019. URL: http://maude.lcc.uma.es/BPMN-R/ *Cited on page 59.*

[173] Francisco Durán, Camilo Rocha, and Gwen Salaün. "Symbolic Specification and Verification of Data-aware BPMN Processes using Rewriting Modulo SMT". In: *12th International Workshop on Rewriting Logic and its Applications, WRLA*. 2018, pp. 76–97. DOI: 10.1007/978-3-319-99840-4_5 *Cited on pages 59, 60, 63, 153.*

[174] Francisco Durán, Camilo Rocha, and Gwen Salaün. *BPMN-SMT*. 2019. URL: http://maude.lcc.uma.es/BPMN-SMT *Cited on page 59.*

[175] Mayssa Bessifi. *BPMN2EventB*. 2021. URL: https://drive.google.com/drive/folders/1wgZCEP1tuc079aJ3guqnSJRU70HQcyp1 *Cited on page 59.*

[176] Emanuele De Angelis et al. *VeriMAP*. 2014. URL: http://map.uniroma2.it/VeriMAP/ *Cited on page 59.*

[177] Wil M. P. van der Aalst et al. "Soundness of Workflow Nets: Classification, Decidability, and Analysis". In: *Formal Aspects of Computing* 23 (2011), pp. 333–363. DOI: 10.1007/s00165-010-0161-4 *Cited on pages 58, 94.*

References for Chapter 4: BPMN and Communication

[1] Marlon Dumas et al. *Fundamentals of Business Process Management, Second Edition*. Springer, 2018 *Cited on pages 1, 10, 11, 94, 95.*

[3] OMG Group. *Business Process Modeling Notation*. 2013. URL: http://www.omg.org/spec/BPMN/2.0.2/ *Cited on pages 1, 3, 4, 12, 20, 21, 58, 72, 90, 91, 99, 109.*

[23] Flavio Corradini et al. "A Classification of BPMN Collaborations based on Safeness and Soundness Notions". In: *Proceedings of the 25th International Workshop on Expressiveness in Concurrency,EPTCS*. 2018, pp. 37–52. DOI: 10.4204/EPTCS.276.5 *Cited on pages 15, 17, 94–96, 132, 138, 150, 151.*

[39] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Communication ACM* 12 (1969), pp. 576–580. DOI: 10.1145/1562764.1562779 *Cited on pages 27, 80.*

[81] Carlo Combi, Barbara Oliboni, and Francesca Zerbato. "A Modular Approach to the Specification and Management of Time Duration Constraints in BPMN". In: *Information Systems* 84 (2019), pp. 111–144. DOI: 10.1016/j.is.2019.04.010 *Cited on pages 47, 54, 57, 94, 150, 152.*

[177] Wil M. P. van der Aalst et al. “Soundness of Workflow Nets: Classification, Decidability, and Analysis”. In: *Formal Aspects of Computing* 23 (2011), pp. 333–363. DOI: 10.1007/s00165-010-0161-4 *Cited on pages 58, 94.*

[178] Florent Chevrou, Aurélie Hurault, and Philippe Quéinnec. “A Modular Framework for Verifying Versatile Distributed Systems”. In: *International Conference on High Performance Computing & Simulation, HPCS.* 2018. DOI: 10.1109/HPCS.2018.00121 *Cited on page 68.*

[179] Florent Chevrou, Aurélie Hurault, and Philippe Quéinnec. “On the Diversity of Asynchronous Communication”. In: *Formal Aspects of Computing* 28 (2016), pp. 847–879. DOI: 10.1007/s00165-016-0379-x *Cited on pages 74, 131.*

[180] Leslie Lamport. “Time, Clocks and the Ordering of Events in a Distributed System”. In: *Communications of the ACM* 21 (1978), pp. 558–565. DOI: 10.1145/359545.359563 *Cited on page 77.*

[181] Reinhard Schwarz and Friedemann Mattern. “Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail”. In: *Distributed Computing* 7 (1994), pp. 149–174. DOI: 10.1007/BF02277859 *Cited on page 77.*

[182] Ajay D. Kshemkalyani and Mukesh Singhal. “Necessary and Sufficient Conditions on Information for Causal Message Ordering and Their Optimal Implementation”. In: *Distributed Computing* 11 (1998), pp. 91–111. DOI: 10.1007/s004460050044 *Cited on page 77.*

[183] Michel Raynal, André Schiper, and Sam Toueg. “The Causal Ordering Abstraction and a Simple Way to Implement It”. In: *Information Processing Letters* 39 (1991), pp. 343–350. DOI: 10.1016/0020-0190(91)90008-6 *Cited on page 77.*

[184] Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. “Synchronous, Asynchronous, and Causally Ordered Communication”. In: *Distributed Computing* 9 (1996), pp. 173–191. DOI: 10.1007/s004460050018 *Cited on page 78.*

[185] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking.* MIT Press, 2001 *Cited on page 78.*

[186] David L. Dill et al. “Protocol Verification as a Hardware Design Aid”. In: *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computer & Processors, ICCD.* 1992, pp. 522–525 *Cited on page 78.*

[187] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia and al. *NuSMV: a new symbolic model checker.* 2010. URL: https://nusmv.fbk.eu *Cited on page 78.*

[188] Clifford B. Jones. *Systematic software development using VDM (2. Edition).* Prentice Hall, 1991 *Cited on page 80.*

[189] John V. Guttag et al. *Larch: Languages and Tools for Formal Specification.* Springer, 1993 *Cited on page 80.*

[190] J. Michael Spivey. *Z Notation - a reference manual (2.nd Edition).* Prentice Hall, 1992 *Cited on page 80.*

[191] Wil M. P. van der Aalst. “Verification of Workflow Nets”. In: *International Conference on Application and Theory of Petri Nets, ICATPN.* 1997, pp. 407–426. DOI: 10.1007/3-540-63139-9_48 *Cited on pages 94, 95.*

References for Chapter 5: BPMN and Time

[3] OMG Group. *Business Process Modeling Notation.* 2013. URL: http://www.omg.org/spec/BPMN/2.0.2/ *Cited on pages 1, 3, 4, 12, 20, 21, 58, 72, 90, 91, 99, 109.*

[15] Andreas Lanz, Barbara Weber, and Manfred Reichert. "Time patterns for process-aware information systems". In: *Requirements Engineering* 19 (2014), pp. 113–141. DOI: 10.1007/s00766-012-0162-3 *Cited on pages 3, 99, 114, 115, 117, 118, 120, 122, 123.*

[16] *ISO 8601:2004, Data elements and interchange formats — Information interchange — Representation of dates and times.* Standard. ISO, 2004 *Cited on pages 4, 99.*

[192] Christopher Alexander. *The Timeless Way of Building.* Oxford University Press, 1979 *Cited on page 114.*

[193] Workflow Patterns Initiative. *Workflow Patterns home page.* 2017. URL: http://www.workflowpatterns.com/ *Cited on page 114.*

[194] Arthur H. M. ter Hofstede et al. *Modern Business Process Automation - YAWL and its Support Environment.* Springer, 2010 *Cited on page 114.*

[195] Hai Zhuge, To-yat Cheung, and Hung-keng Pung. "A Timed Workflow Process Model". In: *Journal of Systems and Software* 55 (2001), pp. 231–243. DOI: 10.1016/S0164-1212(00)00073-X *Cited on page 114.*

[196] Wil M. P. van der Aalst, Maja Pesic, and Minseok Song. "Beyond Process Mining: From the Past to Present and Future". In: *22nd International Conference on Advanced Information Systems Engineering, CAiSE.* 2010, pp. 38–52. DOI: 10.1007/978-3-642-13094-6_5 *Cited on page 114.*

[197] Johann Eder, Euthimios Panagos, and Michael Rabinovich. "Time Constraints in Workflow Systems". In: *11th International Conference Advanced Information Systems Engineering, CAiSE.* 1999, pp. 286–300. DOI: 10.1007/3-540-48738-7_22 *Cited on page 114.*

[198] Carlo Combi et al. "Conceptual Modeling of Temporal Clinical Workflows". In: *14th International Symposium on Temporal Representation and Reasoning, TIME.* 2007, pp. 70–81. DOI: 10.1007/978-3-642-13094-6_5 *Cited on page 114.*

[199] Carlo Combi et al. "Conceptual Modeling of Flexible Temporal Workflows". In: *ACM Transactions on Autonomous and Adaptive Systems* 7 (2012), 19:1–19:29. DOI: 10.1145/2240166.2240169 *Cited on page 114.*

[200] Nick Russell, Wil M. P. van der Aalst, and Arthur H. M. ter Hofstede. *Workflow Patterns: The Definitive Guide.* MIT Press, 2016 *Cited on page 121.*

References for Chapter 6: fbpmn: Formal BPMN Framework

[17] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison Wesley, 2002 *Cited on pages 4, 27, 28, 127.*

[23] Flavio Corradini et al. "A Classification of BPMN Collaborations based on Safeness and Soundness Notions". In: *Proceedings of the 25th International Workshop on Expressiveness in Concurrency,EPTCS.* 2018, pp. 37–52. DOI: 10.4204/EPTCS.276.5 *Cited on pages 15, 17, 94–96, 132, 138, 150, 151.*

[46] Mana Taghdiri and Daniel Jackson. "A Lightweight Formal Analysis of a Multicast Key Management Scheme". In: *Formal Techniques for Networked and Distributed Systems - FORTE 2003, 23rd IFIP WG 6.1 International Conference, 2003, Proceedings.* 2003, pp. 240–256. DOI: 10.1007/978-3-540-39979-7_16 *Cited on pages 34, 127.*

[47] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, 2012 *Cited on pages 34, 36, 38, 132.*

[108] Ajay Krishna, Pascal Poizat, and Gwen Salaün. "Checking Business Process Evolution". In: *Science of Computer Programming* 170 (2019), pp. 1–26. DOI: 10.1016/j.scico.2018.09.007 *Cited on pages 49, 54, 138.*

[138] Flavio Corradini et al. "Animating Multiple Instances in BPMN Collaborations: From Formal Semantics to Tool Support". In: *16th International Conference on Business Process Management, BPM.* 2018, pp. 83–101. DOI: 10.1007/978-3-319-98648-7_6 *Cited on pages 52, 54, 59, 60, 63, 138, 150, 151, 153.*

[140] Flavio Corradini et al. "Well-structuredness, Safeness and Soundness: A Formal Classification of BPMN Collaborations". In: *Journal of Logical and Algebraic Methods in Programming* 119 (2021), p. 100630. DOI: 10.1016/j.jlamp.2020.100630 *Cited on pages 52, 54, 61, 128.*

[179] Florent Chevrou, Aurélie Hurault, and Philippe Quéinnec. "On the Diversity of Asynchronous Communication". In: *Formal Aspects of Computing* 28 (2016), pp. 847–879. DOI: 10.1007/s00165-016-0379-x *Cited on pages 74, 131.*

[201] Eclipse Foundation. *Eclipse BPMN2 Modeler.* 2021. URL: https://www.eclipse.org/bpmn2-modeler/ *Cited on page 128.*

[202] Camunda services GmbH. *Camunda v.7.7.0.* 2018. URL: http://www.camunda.com *Cited on page 128.*

[203] Signavio GmbH. *Signavio Home Page.* 2021. URL: http://www.signavio.com/ *Cited on page 128.*

[204] Florent Chevrou, Aurélie Hurault, and Philippe Quéinnec. "A Modular Framework for Verifying Versatile Distributed Systems". In: *Journal of Logical and Algebraic Methods in Programming* 108 (2019), pp. 24–46. DOI: 10.1016/j.jlamp.2019.05.008 *Cited on page 131.*

[205] Rim Saddem-Yagoubi, Pascal Poizat, and Sara Houhou. "Business Processes Meet Spatial Concerns: the sBPMN Verification Framework". In: *24th International Symposium on Formal Methods (FM).* 2021 *Cited on pages 142, 153.*

[206] Pardi project team. *BPMN Formalization.* 2019. URL: http://vacs.enseeiht.fr/bpmn/ *Cited on page 144.*

References for Chapter 7: Conclusion

[14] Saoussen Cheikhrouhou et al. "Toward a Time-centric modeling of Business Processes in BPMN 2.0". In: *The 15th International Conference on Information Integration and Web-based Applications & Services, IIWAS.* 2013, pp. 154–163. DOI: 10.1145/2539150.2539182 *Cited on pages 3, 152.*

[23] Flavio Corradini et al. "A Classification of BPMN Collaborations based on Safeness and Soundness Notions". In: *Proceedings of the 25th International Workshop on Expressiveness in Concurrency,EPTCS.* 2018, pp. 37–52. DOI: 10.4204/EPTCS.276.5 *Cited on pages 15, 17, 94–96, 132, 138, 150, 151.*

[64] Remco M Dijkman, Marlon Dumas, and Chun Ouyang. "Semantics and Analysis of Business Process Models in BPMN". In: *Information and Software technology* 50 (2008), pp. 1281–1294. DOI: 10.1016/j.infsof.2008.02.006 *Cited on pages 46, 47, 54–57, 59, 60, 62, 150, 151.*

[66] Eindhoven University of Technology Process Mining Group. *Process Mining Framework (PROM).* 2020. URL: http://www.processmining.org/prom/start *Cited on pages 46, 153.*

[71] Pieter Van Gorp and Remco M. Dijkman. “A Visual Token-based Formalization of BPMN 2.0 based on In-place Transformations”. In: *Information and Software Technology* 55 (2013), pp. 365–394. DOI: 10.1016/j.infsof.2012.08.014 *Cited on pages 46, 54, 55, 57–60, 62, 150, 151, 153.*

[81] Carlo Combi, Barbara Oliboni, and Francesca Zerbato. “A Modular Approach to the Specification and Management of Time Duration Constraints in BPMN”. In: *Information Systems* 84 (2019), pp. 111–144. DOI: 10.1016/j.is.2019.04.010 *Cited on pages 47, 54, 57, 94, 150, 152.*

[86] Ahmed Kheldoun, Kamel Barkaoui, and Malika Ioualalen. “Formal Verification of Complex Business Processes Based on High-level Petri Nets”. In: *Information Sciences* 385 (2017), pp. 39–54. DOI: 10.1016/j.ins.2016.12.044 *Cited on pages 47, 54, 57, 59, 60, 62, 150, 151.*

[89] Chanon Dechsupa, Wiwat Vatanawood, and Arthit Thongtak. “Transformation of the BPMN Design Model into a Colored Petri Net using the Partitioning Approach”. In: *IEEE Access* 6 (2018), pp. 38421–38436. DOI: 10.1109/ACCESS.2018.2853669 *Cited on pages 48, 54, 57, 59, 60, 62, 150, 151.*

[95] JianHong Ye et al. “Formal Semantics of BPMN Process Models using YAWL”. In: *Second International Symposium on Intelligent Information Technology Application.* 2008, pp. 70–74. DOI: 10.1109/IITA.2008.68 *Cited on pages 48, 54, 57, 59, 60, 62, 150, 151.*

[99] Kenji Watahiki, Fuyuki Ishikawa, and Kunihiko Hiraishi. “Formal Verification of Business Processes with Temporal and Resource Constraints”. In: *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, SMC.* 2011, pp. 1173–1180. DOI: 10.1109/ICSMC.2011.6083857 *Cited on pages 49, 54, 55, 57, 150, 152.*

[104] Saoussen Cheikhrouhou et al. “Enhancing Formal Specification and Verification of Temporal Constraints in Business Processes”. In: *IEEE International Conference on Services Computing, SCC.* 2014, pp. 701–708. DOI: 10.1109/SCC.2014.97 *Cited on pages 49, 54, 55, 57, 150, 152.*

[116] Peter YH Wong and Jeremy Gibbons. “A Relative Timed Semantics for BPMN”. In: *Electronic Notes in Theoretical Computer Science* 229 (2009), pp. 59–75. DOI: 10.1016/j.entcs.2009.06.029 *Cited on pages 50, 54, 55, 57, 150, 152.*

[117] Peter Wong YH and Jeremy Gibbons. “Formalisations and Applications of BPMN”. In: *Science of Computer Programming* 76 (2011), pp. 633–650. DOI: 10.1016/j.scico.2009.09.010 *Cited on pages 50, 54, 55, 57, 150, 151.*

[119] Manuel I. Capel and Luis Eduardo Mendoza. “Automating the Transformation from BPMN Models to CSP+ T Specifications”. In: *35th Annual IEEE Software Engineering Workshop, SEW.* 2012, pp. 100–109. DOI: 10.1109/SEW.2012.17 *Cited on pages 50, 54, 55, 57, 59, 60, 63, 150, 152.*

[125] Vitus SW Lam. “A Precise Execution Semantics for BPMN”. In: *International Journal of Computer Science* 39 (2012), pp. 20–33. DOI: 10.1109/MIC.2004.58 *Cited on pages 51, 54, 57, 150, 151.*

[128] Nissreen A. S. El-Saber and Artur Boronat. “BPMN Formalization and Verification using Maude”. In: *Proceedings of the 2014 Workshop on Behaviour Modelling - Foundations and Applications, BM-FA.* 2014, pp. 1–12. DOI: 10.1145/2630768.2630769 *Cited on pages 51, 54, 55, 57, 150, 151.*

[132] Flavio Corradini et al. “A Formal Approach to Modelling and Verification of Business Process Collaborations”. In: *Science of Computer Programming* 166 (2018), pp. 35–70. DOI: 10.1016/j.scico.2018.05.008 *Cited on pages 51, 54, 55, 57, 59, 60, 63, 150, 151.*

[137] Apromore Pty Ltd. *Apromore*. 2020. URL: https://apromore.org *Cited on pages 52, 153.*

[138] Flavio Corradini et al. "Animating Multiple Instances in BPMN Collaborations: From Formal Semantics to Tool Support". In: *16th International Conference on Business Process Management, BPM*. 2018, pp. 83–101. DOI: 10.1007/978-3-319-98648-7_6 *Cited on pages 52, 54, 59, 60, 63, 138, 150, 151, 153.*

[142] Francisco Durán and Gwen Salaün. "Verifying Timed BPMN Processes Using Maude". In: *19th IFIP International Conference on Coordination Models and Languages, , COORDINATION*. 2017, pp. 219–236. DOI: 10.1007/978-3-319-59746-1_12 *Cited on pages 52, 54, 55, 57, 59–61, 63, 150, 152.*

[143] Francisco Durán, Camilo Rocha, and Gwen Salaün. "Stochastic analysis of BPMN with time in rewriting logic". In: *Science of Computer Programming* 168 (2018), pp. 1–17. DOI: 10.1016/j.scico.2018.08.007 *Cited on pages 52, 54, 55, 57, 59–61, 63, 150, 152.*

[173] Francisco Durán, Camilo Rocha, and Gwen Salaün. "Symbolic Specification and Verification of Data-aware BPMN Processes using Rewriting Modulo SMT". In: *12th International Workshop on Rewriting Logic and its Applications, WRLA*. 2018, pp. 76–97. DOI: 10.1007/978-3-319-99840-4_5 *Cited on pages 59, 60, 63, 153.*

[205] Rim Saddem-Yagoubi, Pascal Poizat, and Sara Houhou. "Business Processes Meet Spatial Concerns: the sBPMN Verification Framework". In: *24th International Symposium on Formal Methods (FM)*. 2021 *Cited on pages 142, 153.*

[207] Andreas Lanz, Barbara Weber, and Manfred Reichert. "Workflow Time Patterns for Process-Aware Information Systems". In: *11th International Workshop Enterprise, Business-Process and Information Systems Modeling, BPMDS*. 2010, pp. 94–107. DOI: 10.1007/978-3-642-13051-9_9 *Cited on page 150.*

[208] Luis E. Mendoza Morales, Manuel I. Capel Tuñón, and Mariéa A. Pérez. "AA Formalization Proposal of Timed BPMN for Compositional Verification of Business Processes". In: *12th International Conference on Enterprise Information Systems, ICEIS*. 2010, pp. 388–403. DOI: 10.1007/978-3-642-19802-1_27 *Cited on pages 150, 152.*

[209] Andreas Lanz, Manfred Reichert, and Barbara Weber. "Process Time Patterns: A Formal Foundation". In: *Information Systems* 57 (2016), pp. 38–68. DOI: 10.1016/j.is.2015.10.002 *Cited on page 152.*

[210] Camunda Inc. *BPMN 2.0 Symbol Reference*. 2020. URL: https://camunda.com/bpmn/examples/ *Cited on page 152.*

[211] Huu Nghia Nguyen, Pascal Poizat, and Fatiha ZaıÜdi. "A Symbolic Framework for the Conformance Checking of Value-Passing Choreographies". In: *10th International Conference on Service-Oriented Computing, ICSOC*. 2012, pp. 525–532. DOI: 10.1007/978-3-642-34321-6_36 *Cited on page 153.*

[212] Yuliang Li, Alin Deutsch, and Victor Vianu. "VERIFAS: A Practical Verifier for Artifact Systems". In: *VLDB Endowment Inc.* 11 (2017), pp. 283–296. DOI: 10.14778/3157794.3157798 *Cited on page 153.*

[213] Diego Calvanese et al. "Formal Modeling and SMT-Based Parameterized Verification of Data-Aware BPMN". In: *17th International Conference on Business Process Management, BPM*. 2019, pp. 157–175. DOI: 10.1007/978-3-030-26619-6_12 *Cited on page 153.*

[214] Silvio Ghilardi et al. "Petri Nets with Parameterised Data- Modelling and Verification". In: *18th International Conference on Business Process Management, BPM*. 2020, pp. 55–74. DOI: 10.1007/978-3-030-58666-9_4 *Cited on page 153.*