THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Laboratoire d'Informatique de Grenoble

Analyse, optimisation et débogage de processus BPMN

Analysis, optimisation, and debugging of BPMN processes

Présentée par :

Quentin NIVON

Direction de thèse :

Gwen SALAUN PROFESSEUR DES UNIVERSITES, UNIVERSITE GRENOBLE ALPES	Directeur de thèse

Rapporteurs :

MASSIMO MECELLA
FULL PROFESSOR, UNIV. DEGLI STUDI DI ROMA LA SAPIENZA
PASCAL POIZAT
PROFESSEUR DES UNIVERSITES, UNIVERSITE PARIS NANTERRE

Thèse soutenue publiquement le **12 décembre 2025**, devant le jury composé de :

CLAUDIA RONCANCIO, PROFESSEURE DES UNIVERSITES, GRENOBLE INP - UGA	Présidente
GWEN SALAUN, PROFESSEUR DES UNIVERSITES, UNIVERSITE GRENOBLE ALPES	Directeur de thèse
MASSIMO MECELLA, FULL PROFESSOR, UNIV. DEGLI STUDI DI ROMA LA SAPIENZA	Rapporteur
PASCAL POIZAT, PROFESSEUR DES UNIVERSITES, UNIVERSITE PARIS NANTERRE	Rapporteur
OLIVIER BARAIS, PROFESSEUR DES UNIVERSITES, UNIVERSITE DE RENNES	Examinateur
REMCO DIJKMAN, FULL PROFESSOR, TECHNISCHE UNIVERSITEIT EINDHOVEN	Examinateur

Remerciements

En premier lieu, je tiens à remercier mon directeur de thèse, Gwen SALAÜN, pour son encadrement pendant ces 3 années (et demies, puisqu'il me supportait déjà lors de mon stage de Master 2), et tout particulièrement pour le soin et le temps qu'il a consacré aux multiples relectures d'articles, révisions de présentations, et discussions parfois interminables autour de sujets scientifiques comme personnels, sérieux ou plus incongrus. Pour cela, merci.

Ensuite, j'aimerais remercier chaleureusement les membres de l'équipe CONVECS pour leur accueil, leur gentillesse, leur aide et le partage de leur expérience, ainsi que tous les fous rires à la cafèt', en séminaires, ou même en coup de vent dans les couloirs de l'Inria. En particulier, j'aimerais remercier Frédéric LANG, pour la patience dont il a fait preuve à mon égard, afin de répondre de manière détaillée à mes innombrables questions, pour nos différents échanges dans le cadre du cours de Sémantique des Langages de Programmation et Compilation, pour son travail de programmation et de relecture dans le cadre de notre contribution commune sur la vérification de processus, ainsi que pour son humour (parfois douteux...), sa bonne humeur contagieuse, et toutes ses blagues vaseuses. Pour tout cela, merci. J'aimerais également remercier Hubert GARAVEL, pour son aide dans la compréhension du langage LNT, ainsi que pour son accompagnement et sa supervision du travail effectué au sein du model board du Model Checking Contest, et enfin pour ses remarques acérées et ses blagues épicées dont le détail ne saurait être divulgué dans ce document. Je souhaiterais de plus remercier mon directeur d'équipe, Radu MATEESCU, qui, au-delà de m'avoir motivé par ses nombreuses méthodes traditionnelles roumaines, a su m'enseigner quelques rudiments de logique temporelle. Je n'oublierai pas non plus ses anecdotes savoureuses, relatées autour d'un café pris dans une position accroupie tout aussi mémorable. Enfin, je voudrais remercier Wendelin SERWE. Bien qu'étant moins nombreux qu'avec les autres membres de l'équipe, nos contacts ont toujours été utiles, humainement comme scientifiquement. Ces quelques moments passés ensemble m'ont tout de même permis de découvrir une caractéristique notable de Wendelin : son amour pour la nourriture, qui, bien que non encore démontré, s'est toujours révélé empiriquement vrai. Aux membres permanents de l'équipe CONVECS, un grand merci.

Pour finir, je voudrais remercier tous ceux qui, comme moi, font ou ont fait partie de cette équipe pour une durée déterminée : les doctorants et post-doctorants. A ceux qui m'ont vu arriver, que j'ai vu devenir des docteurs accomplis et qui sont partis avant moi, un grand merci. Merci donc à Irman, Ahang, Pierre, Philippe, Jean-Baptiste et Lucie d'avoir ponctué mon passage de suggestions, de remarques, de fous rires, et de tant d'autres choses encore. A vous, je dis merci.

Pour ceux qui, à l'inverse, me verront partir tandis que je les ai vus arriver, je vous souhaite le meilleur pour venir à bout de ce périple. Je pense pouvoir dire que l'équipe CONVECS accueille ses nouveaux membres comme on accueille un nouvel arrivant dans sa famille : avec chaleur et bienveillance. En d'autres termes, je n'aurais pas pu rêver meilleure équipe. Je suis certain que la gentillesse, la sagacité et la rigueur de ses membres ont été autant

de clés me permettant de mener à bout cette thèse dans les meilleures conditions.

Au-delà du cadre professionnel, j'aimerais remercier ma compagne Pauline pour son soutien dans les moments de doute, moments dans lesquels il est parfois difficile d'entrevoir la lumière ou d'entendre raison. Pour ton accompagnement dans ces moments, je te remercie profondément. J'aimerais également remercier famille et amis, qui ont su me rappeler au travers de questions toutes plus pertinentes les unes que les autres ("c'est sur quoi déjà ton stage ?", "rappelle-moi le sujet de ton truc que tu fais là", "ça va te faire combien d'années après le bac tout ça ?", etc.) à quel point mon travail les intéressait et combien ils étaient désireux d'en apprendre plus à son sujet. A vous tous qui m'avez épaulé, je dis merci.

Enfin et surtout, je souhaiterais remercier mes parents, pour l'éducation qu'ils m'ont donnée, pour m'avoir insufflé le goût de l'informatique, pour m'avoir permis d'aller si loin dans cette voie, et pour m'avoir toujours poussé à donner le meilleur de moi-même. Maman, je n'oublierai pas ta compréhension approfondie de mon sujet de thèse (si tu te poses la question, oui, tu es bien l'autrice d'au moins une des questions pertinentes mentionnées plus haut...), ni l'intérêt que tu as porté à mon travail pendant ces trois années, notamment pendant mes jours de télétravail à la maison, où tu as su me montrer combien la notion de télétravail relevait plus de la télé que du travail à tes yeux. Papa, bien que tu ne puisses jamais lire ces lignes, j'espère que tu es fier de moi, et du chemin que j'ai parcouru depuis l'obtention de ma licence d'informatique jusqu'à aujourd'hui. Je sais qu'en relisant ce manuscrit, tu m'aurais gratifié d'un "mouais, t'aurais pu faire mieux", mais que c'était ta façon à toi de communiquer, et que ça m'aurait amplement suffi. Il y a sans doute un peu de toi dans ces lignes, donc, pour cette aide inconsciente, merci.

Abstract

Modelling and designing business processes have become crucial activities for companies in the last decades. Consequently, multiple workflow modelling notations emerged. Among them, the Business Process Modelling Notation (BPMN) is now considered as the *de facto* standard for process modelling. The BPMN notation requires a certain level of expertise to allow one to write correct and well-structured processes compliant with some expected requirements. The BPMN modelling phase can thus become tedious, and even error-prone if carried out by non-experts. The first part of this thesis consists in providing a solution to help users modelling BPMN processes. To achieve this goal, the proposed approach takes as input the requirements of the user in a textual format, in which the tasks and their ordering constraints are informally described. It then makes use of Large Language Models (LLMs) to translate these constraints into a machine-readable format. From this internal format, the approach generates and returns a BPMN process satisfying these constraints. As a side contribution, this thesis also presents techniques to verify that a process does not deviate from its expected behaviour. Such verifications are usually performed using classical model checking techniques. However, we thought that a user not familiar with the BPMN notation would struggle using such techniques, and more precisely, writing the expected behaviour of the process in the form of temporal logic properties. Thus, the core of this side contribution consists in facilitating the writing of such properties by allowing the user to generate them directly from their textual description.

Recent studies suggested that, once built, business processes are subject to changes throughout their lifetime. In companies, such changes may often lead to non-optimal processes, responsible of issues such as the increase of the execution time, or of the costs related to them. To be able to optimise processes, there is a need to have at hand an explicit model of their behavior and quantitative features. Thus, beside the processes themselves, usually modelled using workflow-based notations, one must provide, among others, an explicit description of the durations and the resources required by the tasks composing these processes. The second part of this thesis consists in providing a solution based on *refactoring techniques*, whose goal is to change the structure of the process in order to optimise one or several criteria of interest, such as process execution time, resource usage, or total costs. To do so, the proposed approaches consist of various ingredients. For instance, we propose *refactoring patterns*, useful for moving the tasks of the process from one place to another while partially preserving its semantics. We also make use of *simulation techniques* to compute metrics of interest from the execution of (one or several instances of) the process. Similarly, we utilise several *exploration algorithms* in order to navigate through the space of solutions. In the end, the multiple presented approaches all return an optimised version of the original process given as input.

Résumé

Durant les dernières décennies, la modélisation et la conception de processus métier sont devenues des activités cruciales pour de nombreuses sociétés. En conséquence, de multiples notations permettant de modéliser ces processus ont émergé, majoritairement basées sur les flux de travaux. Parmi elles, la méthode appelée *notation et modèle de processus métier* (*Business Process Model and Notation* (BPMN), en anglais) est aujourd'hui considérée comme le standard pour la modélisation de processus. La notation BPMN requiert un certain niveau d'expertise pour permettre à ses utilisateurs d'écrire des processus corrects, bien structurés, et conformes aux exigences. La phase de modélisation peut donc devenir fastidieuse, et même sujette à erreurs si elle n'est pas effectuée par des experts de la notation. La première partie de cette thèse consiste à fournir une solution visant à aider les utilisateurs à modéliser des processus BPMN. Pour ce faire, l'approche proposée prend en entrée les exigences de l'utilisateur écrites dans un format textuel, dans lesquelles les tâches et les contraintes les ordonnant sont décrites informellement. Ensuite, l'approche utilise des *grands modèles de langage* (*Large Language Models* (LLMs), en anglais) pour traduire ces contraintes dans un format compréhensible par une machine. A partir de ce format interne, l'approche génère et retourne un processus BPMN satisfaisant les contraintes de départ. Cette thèse présente également, comme contribution secondaire, des techniques permettant de vérifier qu'un processus ne dévie pas de son comportement attendu. De telles vérifications sont généralement effectuées grâce à des techniques classiques de vérification de modèle. Or, nous avons pensé qu'un utilisateur n'étant pas familier avec la notation BPMN pourrait rencontrer des difficultés dans l'utilisation de telles techniques, et plus particulièrement, dans l'écriture des propriétés de logique temporelle décrivant le comportement attendu. Ainsi, le cœur de cette contribution secondaire consiste à faciliter l'écriture de telles propriétés en permettant à l'utilisateur de les générer directement à partir de leur description textuelle.

De récentes études suggèrent que, une fois conçus, les processus métiers évoluent, et ce tout au long de leur cycle de vie. Dans les sociétés, de tels changements peuvent parfois amener à des processus non-optimaux, responsables de problèmes tels que l'augmentation du temps d'exécution ou des coûts qui leur sont liés. Afin d'optimiser ces processus, il est nécessaire d'avoir à portée de main un modèle décrivant explicitement leur comportement, ainsi que les aspects quantitatifs qui y sont liés. Ainsi, au-delà des processus eux-mêmes, généralement modélisés dans une notation basée sur les flux de travaux, il est nécessaire de fournir, entre autres, une description explicite des durées et des ressources requises par les tâches composant ces processus. La seconde partie de cette thèse consiste à fournir une solution basée sur des *techniques de refactorisation*, dont le but est de changer la structure du processus afin d'optimiser un ou plusieurs critères, tels que le temps d'exécution, l'utilisation des ressources, ou les coûts totaux. Pour ce faire, les approches proposées sont composées de plusieurs ingrédients. Par exemple, nous présentons des *patrons de refactorisation*, utiles pour déplacer les tâches du processus d'un endroit à un autre tout en préservant partiellement sa sémantique. Nous utilisons également des *techniques de simu-*

lation pour calculer des métriques, basées sur l'exécution d'une ou plusieurs instances du processus. De même, nous détaillons plusieurs *algorithmes d'exploration* ayant pour objectif de naviguer à travers l'espace de solutions. Au final, les multiples approches présentées retournent toutes une version optimisée du processus donné en entrée.

Contents

Chapter 1

Introduction

1.1 Context

According to history, the very first representation of a business process was proposed by the economist Adam Smith in 1776, who described the execution flow of a pin factory [Smi76]. He also introduced the notion of labour division, which consists in dividing a production process into several small tasks, each of which will be performed by a specialised worker. These two notions sealed the basis of business processes as they are known nowadays. In the 20th century, Frederick Winslow Taylor's ideas were seen has a revolution in the field, as he proposed a standardisation of the processes, and a clear definition of the roles involved in the business processes [Tay11]. Later in that century, several people tried to provide a global definition of business processes [RB90, Dav93, HC93, JMP93], each focusing on different aspects of the process, with more or less details. This desire to provide a rigorous, unified definition of business processes paved the way to the creation of a new discipline: the *business process management*.

Business Process Management (BPM) [JN06, Wes07, Pan12] is a holistic discipline encompassing all the fields related to business processes, such as business process discovery, modelling, analysis, measurement, improvement, optimisation or automation [RvSK14]. Since its apparition and its democratisation, it has become widely used in companies [MR08, Neu09], independently of their domains of application. The term *business process modelling* was coined in the 1960s by Stanley Williams [CW71] with the idea of representing business processes as physical control systems in order to better understand them. However, modelling business processes has been a topic of interest since the early 1900's, with the apparition of several modelling notations such as flowcharts [GG21], functional flow block diagrams (FFBDs), control-flow diagrams (CFDs), Gantt charts [Gan10], PERT diagrams [MRCF59], or IDEF [ide98]. More recently, other notations emerged, such as the Unified Modelling Language (UML) [BJR17] and the Business Process Management Notation (BPMN) [OMG11]. Due to their completeness and understandability, both notations rapidly became widely used worldwide standards. Moreover, further research suggested that BPMN could be more suitable than UML for modelling business processes [Whi04, NK06, Wes07], although it was refuted by following studies [BKO10, Gea12]. However, the seed was planted, and many companies started making use of this notation to model their business processes[1].

[1] `https://camunda.com/blog/2024/07/how-13-businesses-feel-about-bpmn/`

The BPMN notation allows one to depict precisely a business process. However, it does not inherently provide support for quantitative aspects of processes, such as resources or durations. To bypass this limitation, several extensions of the notation emerged in the literature, aiming at providing support for, among others, resources [Gro07, AGMW08, SCV11, BDGP16] and time [GT09, WG09, AERDM16]. Such extensions of the language eventually led to the apparition of techniques aiming at performing quantitative analyses of the processes [OLRR12, HS12, DRS18b]. Companies saw in this enrichment of the notation a novel opportunity for improving their business processes, by optimising them with regards to multiple quantitative criteria. Indeed, optimising business processes is a strategic activity in organisations because of its potential to increase profit margins and reduce operational costs.

Similarly, BPMN does not natively provide tools for verifying the syntactic and/or semantic correctness of the processes. However, assessing and asserting the validity of BPMN processes is essential to ensure that they do not deviate from their expected behaviour. To palliate these flaws, several approaches were proposed to verify whether a BPMN process is syntactically correct regarding the standard [RH07, SM07, CMP+20, BGT20], or semantically correct with regards to some expected behaviour [DDO08, KPS17, BGT20].

1.2 Motivations

With almost six millions matching results on Google Scholar, business process management has been (and is still!) a hot topic in the field of computer science. Moreover, studies showed that, in spite of the raise of graphical modelling tools, modelling business processes has always been a tedious [GV06, KO10] and error-prone [ML04, LF14, RSBR14, Gro22] task. On the one hand, it usually requires a deep understanding of the company's needs, and of its intrinsic way of functioning. On the other hand, it necessitates strong competences in modelling, and in particular, in the context of this thesis, an advanced knowledge of the BPMN notation. Indeed, the BPMN modelling tools allow a lot of freedom in the design of the processes, and usually do not provide integrated solutions for asserting their correctness. Thus, despite following some modelling best practices [MW08, SFS11], a novice user can quite easily design a syntactically and/or semantically incorrect process. Moreover, complex structures—such as unbalanced gateways, or nested loops—may be challenging to handle for inexperienced users.

Consequently, the interest in finding solutions to automatically generate business processes has been growing uninterruptedly [dABTS+18]. Between the early 2000s and the late 2010s, most solutions to this problem were making use of Natural Language Processing (NLP) [Tur50] to extract information from a textual representation of the process, and use it to generate the corresponding (graphical) BPMN process [FMP11, HKW18]. A few others [ISP20, FSZ21] decided to tackle the problem from another angle, considering that business processes could be written in an intermediate, simpler—yet machine readable and understandable—format. However, in both cases, the user often has to give a nudge to

solve some misunderstood requirements, or to obtain/generate the required input format. Some others also gave a try to different input formats, such as a hand-drawn representation of the process [SvdALS23], but this requires the user to have knowledge of the notation. Consequently, none of the aforementioned approaches can be considered interaction-free, in the sense that they all require, at some point, human intervention to complete. It is also worth mentioning process mining [vdA16], which consists in analysing event data and/or logs to infer information about a given system. Such techniques were applied successfully to the BPMN modelling problem [CDGBR16, KBdL+18]. However, they require logs as input, that are, detailed, structured information of the process-to-be.

In 2018, the OpenAI company introduced GPT-1 [RNSS18], the first Generative Pre-Trained Transformer. Four years later, they introduced the ChatGPT website[2], which has been a major breakthrough in the field of artificial intelligence [GtH23, Sza23, SMA24] and a fundamental change in people's everyday life [FS23, Shi24, WM24, AHCN24]. GPT (and more generally, large language models [ZZL+25]) lifted numerous barriers in the field of natural language processing due to its capabilities to understand raw textual formats [SWN25]. In addition to that, it also showed strong skills in text generation [BOQ+25].

This fast emergence led to several research questions, the most recurrent one concerning the reliability of GPT's answers [WCP+23, Chu24, ZLC+24]. Further research showed two main flaws in the current existing AI models: they can not reason [Ark23], and they do not know when they are wrong [SMK23]. Aware of these limitations, and oppositely to most of the recent business process generation approaches [KBSvdA24b, EAA+24], we decided to use LLMs as little as possible, and only for tasks that we were not able to complete with classical algorithms. This design decision allowed us to better comprehend what the LLM was doing wrong, in case of errors from its side, and also how the error was correlated to the given input. Moreover, the utilisation of classical algorithms for a large part of the process generation allowed us to preserve control on the different steps composing this transformation, and to provide strong semantics guarantees to the generated model.

Despite corresponding to the original needs of the companies, several studies suggested that, in practice, business processes were not built once and for all in a monolithic way, but had their own lifecycle consisting of several successive phases, which, if needed, could be repeated [GT98, MKPC14, RHB15]. During the lifetime of a business process, several reasons can motivate the modification of its structure: the addition/deletion of a specific task, the adjustment of the process to consider a new regulation or internal directives, the improvement of the process with respect to one or several quantitative criteria (such as overall execution time or total cost), etc. When designing a process, or when updating it, the quality and the correctness of the process must be preserved. However, this may not be the case if this rewriting/reengineering of the process is achieved manually.

Further research showed that modifying the structure of a process in order to optimise it may be challenging when some quantitative aspects are taken into account. For in-

[2] `https://chatgpt.com/`

stance, the structure of the optimised version of a business process may differ depending on whether the desired optimisation targets the usage of certain resources, the cost of the process, or its execution time. Moreover, applying these modifications manually is nearly impossible due to the complexity of this enriched model. Therefore, there is a need for automated techniques aiming at optimising a business process throughout its lifetime. These research questions opened up the door to multiple proposals, targeting different aspects of optimisation.

Resource optimisation [DRS19, DRS21, FSZ24] is a technique allowing, among others, to reduce the execution time of a business process by modifying the number of replicas of each resource available to the process. This approach does not change the structure of the process, but may be difficult to apply in practice, as it usually requires some flexibility in the budget of the companies. Indeed, changing the number of available resources may require the acquisition of new machines, or the hiring of new employees. On the other hand, approaches such as [GT09] give insights on how scheduling could be used to optimise business processes. Such approaches do not require any change in the structure of the process, nor any budget flexibility. However, they are inherently unsuitable for poorly designed processes, whose structure contain flaws or potential of optimisation. To palliate these drawbacks, several authors [RM05, KL22, DS22] proposed a technique called *process refactoring*, which consists in modifying the structure of a process in order to optimise its execution time. However, in their current form, these approaches only consider a single replica of the resources, a single execution of the considered process, and a single optimisation criterion: the execution time of the process.

1.3 Approach

The approach proposed in this thesis first aims at helping BPMN designers during the modelling phase by providing them a *simple* and *efficient* way of generating *syntactically* and *semantically* correct BPMN processes. The simplest (and most adopted) way of representing a business process for non-expert users is to describe its behaviour textually, in natural language. This format was consequently chosen as input format in our proposal. However, natural language is well-known for its inherent complexity [Lin96] and ambiguity [Jac20]. For this reason, we rely on natural language processing techniques, and, more precisely, on LLMs, to analyse and extract the essential information contained in the description of the process, and transform it into a machine-readable format. Several successive steps are then applied to this output in order to generate the corresponding BPMN process. For the sake of efficiency and correctness, we chose to prompt the LLM once and only once. On the one hand, this unique prompt shortens the execution time of the approach. On the other hand, it reduces the sensitivity and the variability of the result to this single step, while all the remaining ones can ensure strong semantical guarantees.

The LLM prompt contains several information regarding the task that it should perform, the format of its answer, some examples of user prompts and expected answers, and the

description of the process. The answer returned by the LLM is designed to contain the essence of the useful information hidden in the description, that is, the tasks that should compose the process, along with their relationships. This answer is then transmitted to a pipeline consisting of several steps, with the goal of extracting and manipulating all the information that it contains in order to produce a relevant BPMN process. Each of these steps has a specific goal aiming at enriching a previous version of the process with new information. On the one hand, this solution is useful for non-expert users since it provides a way to specify BPMN processes without mastering the intricacies of the notation. On the other hand, it is also helpful for expert users, because it simplifies and fastens the modelling phase by automatically generating BPMN processes, thus avoiding the burden of graphically writing the entire workflow step by step. Finally, the generated process has the main benefit of being semantically correct with regards to the LLM's answer, and syntactically correct by construction.

Despite being correct with regards to the LLM's answer, the generated process may not perfectly reflect the behaviour of the original description given by the user. This may be due to the inherent ambiguity of natural language, but also to an incomprehension or a misunderstanding of the LLM. Thus, there is a need to verify that the generated process' behaviour corresponds to the expectations of the user. Several techniques aiming at assessing the behaviour of a BPMN process already exist in the literature [DDO08, KPS17, BGT20]. However, they usually require knowledge about behavioral specification languages, such as temporal logics [Pnu77, CE82, EH83, MT08], which is, in our opinion, rather unlikely for users not familiar with BPMN. To palliate this, and similarly to the works proposed in [FC23, CHM+23], we proposed an LLM-based technique relying on patterns [DAC99] to translate behavioural properties written in natural language into their corresponding representation in temporal logic. Given this temporal logic property, the behaviour of the BPMN process can be verified, with the help of classical model checking techniques [BK08].

We stated earlier that, once built, business processes were subject to variations due to, for instance, changes in the company's needs, or desires of optimisation. According to the literature, the field of business process optimisation lacked of solutions aiming at efficiently improving a process, without inducing additional costs. To tackle this research problem, we proposed to extend an approach called *process refactoring* [DS22], which consists in changing the structure of a business process in order to optimise it, with regards to one or multiple criteria. Several possibilities were explored, and can roughly be separated in two blocks. The first one consists in statically analysing the input process, and, guided by the results of this analysis, modify its structure in a one-shot manner. Once generated, this new process is evaluated to assess its quality, and may be subject to slight restructuring if needed. The second one was thought oppositely, and consists in making progressive small changes to the structure of the process, by repeatedly moving one of its tasks to another position. In order to analyse the quality of a given process, metrics related to the desired optimisation criteria are computed, with the help of simulation techniques. Unlike the first refactoring approach, the second may generate an important quantity of processes

at each step, which requires techniques for seeking the best solutions among the set of candidates. Finally, it is worth noting that moving tasks in a BPMN process may not preserve the semantics of the original process, if not performed carefully. For this reason, we rely on well-defined refactoring patterns that ensure the preservation of the original trace semantics of the process, modulo some permutations.

1.4 Contributions

The two main contributions of this thesis correspond to the two research approaches presented respectively in Chapters 3 & 4 of this manuscript, focusing on the generation of BPMN processes from their natural language descriptions, and on the optimisation of such processes via refactoring. Another minor contribution of this thesis resides in the generation of temporal logic properties from their natural language descriptions, with the goal of verifying such properties on BPMN processes with the help of model checking. This contribution is detailed in Section 5.2, as it was mostly thought as a tool, called GIVUP, itself presented in this section. It is also worth mentioning that each contribution presented in this thesis is accompanied by its own fully functional toolchain that was used to test and evaluate the approach that it implements. More details about these tools are given in Chapter 5 of this manuscript. Overall, the contributions of this thesis can be summarised as follows:

— An LLM-based approach aiming at extracting the essential information from a textual description of a process, later used to generate automatically a BPMN representation of that process. This process is syntactically correct by construction, and semantically correct with regards to the information returned by the LLM;
— Three approaches for optimising BPMN processes through refactoring, each mitigating some of the shortcomings of the others. All of them make use of a set of refactoring patterns, guiding the refactoring operation through the generation of an optimised version of the process while ensuring the preservation of several semantic properties;
— An approach aiming at generating temporal logic properties from their textual description, based on patterns, in order to provide facilities for the verification of BPMN processes;
— Four tools (one for the modelling and three for the refactoring), consisting of roughly 45k lines of Java code, which were used to embed, test, and evaluate the quality of the four aforementioned approaches.

1.5 Publications

These three years of work led to several publications in renowned international conferences. These publications are listed below[3].

- **Incremental Synchronization of BPMN Models and Documentations by Leveraging Structural Algorithms and LLMs** [CDNS25]*. David Cremer, Benjamin Dalmas, Quentin Nivon, Gwen Salaün. *Proceedings of the 31st International Conference on Cooperative Information Systems (CoopIS'25)*, October 2025, Marbella, Spain.
- **GIVUP: Automated Generation and Verification of Textual Process Descriptions**. Quentin Nivon, Gwen Salaün, Frédéric Lang. *Proceedings of the 33rd International Conference on Foundations of Software Engineering (FSE'25)*, June 2025, Trondheim, Norway.
- **Automated Generation of BPMN Processes from Textual Requirements**. Quentin Nivon, Gwen Salaün. *Proceedings of the 22nd International Conference on Service-Oriented Computing (ICSOC'24)*, December 2024, Tunis, Tunisia.
- **Semi-Automated Refactoring of BPMN Processes**. Quentin Nivon, Gwen Salaün. *Proceedings of the 24th International Conference on Software Quality, Reliability and Security (QRS'24)*, July 2024, Cambridge, England.
- **Automated Repair of Violated Eventually Properties in Concurrent Programs** [FNS24]*. Irman Faqrizal, Quentin Nivon, Gwen Salaün. *Proceedings of the 12th International Conference on Formal Methods in Software Engineering (FormaliSE'24)*, April 2024, Lisbon, Portugal.
- **Refactoring of Multi-Instance BPMN Processes with Time and Resources**. Quentin Nivon, Gwen Salaün. *Proceedings of the 21st International Conference on Software Engineering and Formal Methods (SEFM'23)*, November 2023, Eindhoven, The Netherlands.

1.6 Thesis Structure

The rest of this thesis is organised as follows:

Chapter 2 presents all the background notions and definitions required to understand the rest of this work, with a focus on classical mathematical notations, details on graphs and trees, an in-depth presentation of the BPMN notation, and a high level presentation of model checking.

[3]the asterisk symbol '*' next to the name of a paper indicates that the cited work is not presented in this manuscript, because of its deviation from the original topic of this PhD.

Chapter 3 presents our approach aiming at generating a BPMN process from a natural language description of its behaviour. In particular, it provides a fine-grained focus on the different steps applied to transform the textual description into the corresponding process.

Chapter 4 details our three approaches aiming at optimising a BPMN process by applying refactoring techniques to it. It starts by introducing the shared concepts of all the approaches, and then details the semantic preservation that they ensure, before presenting and comparing/discussing them.

Chapter 5 introduces the tool support we developed around each of the presented approaches, along with the experiments we conducted to evaluate them. It also introduces our GIVUP tool, providing further information on all its building blocks.

Chapter 6 provides an overview of the related work adjacent to those presented in this thesis. More precisely, it focuses on other modelling approaches for BPMN and temporal logics, and on other works dealing with refactoring.

Chapter 7 concludes this thesis and discusses the future work.

Chapter 2

Preliminaries

"We are like dwarfs sitting on the shoulders of giants. We see more, and things that are more distant, than they did, not because our sight is superior or because we are taller than they, but because they raise us up, and by their great stature add to ours."

John of Salisbury

"If I have seen further, it is by sitting on the shoulders of giants."

Isaac Newton

Contents

This chapter introduces several preliminary notions that will be used throughout this thesis.

2.1 Mathematical Notations

The first preliminary notions that we introduce are those considered as *mathematical notations*. In particular, we first define several *logical operators* that will be used throughout this thesis.

Definition 2.1 (Logical Operators)**.** *Let:*

— $\top$ *be the proposition* true*;*
— $\bot$ *be the proposition* false*;*
— $P \vee Q$ *be the* disjunction *of propositions* P *and* Q*;*
— $P \wedge Q$ *be the* conjunction *of propositions* P *and* Q*;*
— $P \Rightarrow Q$ *be the logical* implication*, i.e.,* $(\neg P) \vee Q$*;*
— $P \Leftrightarrow Q$ *be the logical* equivalence*, i.e.,* $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$*;*
— $\varphi \models M$ *be the satisfaction of a property* φ *by a model* M*.*

Then, we introduce some notions regarding the *tuples*.

Definition 2.2 (Tuples)**.** *Let:*

— $(x_1, ..., x_n)$ *be the* tuple of size n *(also called* n-tuple*), consisting of the elements* x_1 *to* x_n*;*
— $U[i]$ *be the* i^{th} element[1] *of the n-tuple* U*;*
— $\texttt{index}(x_i)$ *be the* index *operator, which returns the index of the first occurrence of element* x_i *in the tuple* $(x_1, ..., x_n)$*, that is*

$$\texttt{index}(x_i) \stackrel{\text{def}}{=} \begin{cases} i & \text{if } x_i \in (x_1, ..., x_n) \\ \bot & \text{otherwise} \end{cases}$$

— $U[s : e] \stackrel{\text{def}}{=} (U[s], ..., U[e])$ *be the* slice *operator, which returns a vector containing all the elements of the* n-tuple U *whose indices belong to* $[s...e]$*.*

Let us now present some notations about *sets*.

Definition 2.3 (Sets)**.** *Let:*

— $\emptyset$ *be the* empty set*;*
— $\mathbb{N}$ *be the set of* natural numbers*;*
— $\mathbb{R}$ *be the set of* real numbers*;*
— $\{x_1, ..., x_n\}$ *be the smallest set containing each element* x_1 *to* x_n[2]*;*
— $E \subseteq E'$ *be the predicate* E is a subset of E*', i.e.,* $\forall x \in E,\ x \in E'$*;*
— $E \subset E'$ *be the predicate* E is a strict subset of E*', i.e.,* $(E \neq E') \wedge (E \subseteq E')$*;*
— $E \cap E'$ *be the* intersection *of the sets* E *and* E*', i.e.,* $\{x \in E \mid x \in E'\}$*;*
— $E \cup E'$ *be the* union *of the sets* E *and* E*', i.e.,* $\{x \in E \ \vee \ x \in E'\}$*;*
— $E \setminus E'$ *be the* difference *of the sets* E *and* E*', i.e.,* $\{x \in E \mid x \notin E'\}$*;*
— $|E|$ *be the* cardinal number *of the set* E*;*

[1] In particular, we have: $\forall i \in [1...n],\ (x_1, ..., x_n)[i] = x_i$.
[2] Hence, multiple occurrences of an element are allowed, but ignored, i.e., $\{x, x\} = \{x\}$

— 2^E *be the* power set *of E, i.e.,* $\{X \subseteq E\}$*;*
— 2^E_n *be the* power set of size n *of E, i.e.,* $\{X \in 2^E \mid |X| = n\}$*;*
— $2^E_{n^+}$ *be the* power set of size greater or equal to n *of E, i.e.,* $\{X \in 2^E \mid |X| \geq n\}$*;*
— $2^E_{n^-}$ *be the* power set of size lower or equal to n *of E, i.e.,* $\{X \in 2^E \mid |X| \leq n\}$*;*
— $\texttt{any}(E, n) \stackrel{\text{def}}{=} (e_1, ..., e_n)$ *be a* function returning a n-tuple *containing elements of E chosen non-deterministically*[3]*;*
— $\texttt{any}(E)$ *be a* shorthand *for* $\texttt{any}(E, 1)[0]$*;*
— $\texttt{any}_p(E, n) \stackrel{\text{def}}{=} (e_1, ..., e_n)$ *be a* function returning a n-tuple *containing elements of E chosen probabilistically*[4]*;*
— $\texttt{any}_p(E) \stackrel{\text{def}}{=}$ *be a* shorthand *for* $\texttt{any}_p(E, 1)[0]$*;*
— $\mathfrak{S}(E)$ *be a* function returning the set of all permutations *of elements of E;*
— $E \times E' \stackrel{\text{def}}{=} \{(x, y) \mid (x \in E) \wedge (y \in E')\}$ *be the* cartesian product *of sets E and E′.*

Finally, we present some classical *mathematical functions* and *probability distributions.*

Definition 2.4 (Functions)**.** *Let:*

— $x \mapsto \lfloor x \rfloor$ *be the* floor *function;*
— $x \mapsto \lceil x \rceil$ *be the* ceil *function;*
— $\bigcirc_{i \in [1...n]} f_i(x) = (f_n \circ f_{n-1} \circ ... \circ f_2 \circ f_1)(x)$ *be the* composition *of functions* $f_1, ..., f_n$ *applied to* x*.*

Definition 2.5 (Probability Distributions)**.** *Let:*

— $\mathcal{N}(\mu, \sigma)$ *be the* normal distribution *of mean* $\mu \in \mathbb{R}$ *and standard-deviation* $\sigma \in \mathbb{R}^+$*;*
— $\mathcal{U}_{[a,b]}$ *be the* (constant) uniform distribution *of parameters* $(a, b) \in \mathbb{R}^2$*, with* $a < b$*.*

2.2 Graph

An important part of the research presented in this thesis relies on *graph structures*, and more precisely on *directed (vertex-)labelled graphs.*

Definition 2.6 (Directed (Vertex-)Labelled Graph)**.** *A* directed (vertex-)labelled graph G *is a 3-tuple* $G \stackrel{\text{def}}{=} (V, E, \Sigma)$*, where:*

— V *is a set of vertices;*
— $E \subseteq V \times V$ *is a set of (directed) edges, whose elements are written* $e = (v, v')$ *or, for understandability,* $e = v \to v'$*;*

[3]Note that $(e_1, ..., e_n)$ may contain repeated elements, i.e., $\forall (i, j) \in [1...n]^2,\ i \neq j \nRightarrow e_i \neq e_j$.
[4]Note that this operator supposes that every element e of E has a well-defined probability of appearance. Otherwise, it behaves as the $\texttt{any}(E, n)$ operator.

— Σ *is a set of labels, each of them corresponding to a vertex* $v \in V$*, accessible using the operator* $\sigma : V \to \Sigma$*.*

A simplification considered in this work is that each vertex v of a directed (vertex-)labelled graph $G = (V, E, \Sigma)$ is uniquely labelled, i.e., $\forall v_i, v_j \in V,\ v_i \neq v_j \Rightarrow \sigma(v_i) \neq \sigma(v_j)$. As this thesis solely deals with directed (vertex-)labelled graphs, they will be referred to as graphs in the rest of this manuscript.

When dealing with graphs, it is common to reason on sequences of connected vertices [BW10]. Depending of the form of the sequence, it is either called a *walk*, a *trail*, or a *path* of the graph.

Definition 2.7 ((Directed) Walk)**.** *Let* $G = (V, E, \Sigma)$ *be a graph. A* (directed) walk *of* G *is a (possibly infinite) list* $w_G = (v_1, v_2, ..., v_n, ...) \in V$ *such that* $\forall i \in \mathbb{N} \mid v_{i+1} \in w_G$*,* $v_i \to v_{i+1} \in E$*. The set of all (directed) walks of* G *is noted* $\mathcal{W}_G$*.*

Definition 2.8 ((Directed) Trail)**.** *Let* $G = (V, E, \Sigma)$ *be a graph. A* (directed) trail *of* G *is a walk* $w_G \in \mathcal{W}_G$ *in which all edges are distinct. The set of all (directed) trails of* G *is noted* $\mathcal{T}_G$*.*

Remark 2.1. *As all the edges of a trail must be distinct, a trail is necessarily finite.*

Definition 2.9 ((Directed) Path)**.** *Let* $G = (V, E, \Sigma)$ *be a graph. A* (directed) path *of* G *is a trail* $t_G \in \mathcal{T}_G$ *in which all vertices are distinct. The set of all (directed) paths of* G *is noted* $\mathcal{P}_G$*.*

Remark 2.2. *As a path is a trail, a path is necessarily finite.*

Despite being differentiated in the literature, these three notions are often mixed and confusingly used in place of each others. Thus, even though this thesis only deals with walks, the term *path* will be abusively employed in the rest of this manuscript to design a walk, without any assumption on the distinguishability of the edges or the vertices. We also introduce the operator $\mathcal{P}_G(v)$ which returns the set of all paths of a graph $G = (V, E, \Sigma)$ starting with the vertex $v \in V$. Finally, for simplicity reasons in the following definitions, a path will always be defined as a n-tuple, regardless of its potential infinity.

When a path contains multiple occurrences of the same vertex, it is said to be *cyclic*. Otherwise, it is said to be *acyclic*.

Definition 2.10 (Cyclic & Acyclic Path)**.** *Let* $G = (V, E, \Sigma)$ *be a graph, and let* $\mathcal{P}_G$ *be its set of paths. For all* $p = (v_1, ..., v_n) \in \mathcal{P}_G$*,* p *is said to be* cyclic*, noted* $p_{\bigcirc}$*, if there exist* $i, j \in [1...n], i \neq j$*, such that* $v_i = v_j$*, and* acyclic*, noted* $p_{\not\bigcirc}$*, otherwise.*

Similarly, a graph is *cyclic* when it contains at least one cyclic path, and *acyclic* otherwise.

Remark 2.3. *A cyclic graph* $G = (V, E, \Sigma)$ *necessarily contains infinite paths, that are,* $\{p \in \mathcal{P}_G \mid |p| = \infty\}$*.*

In graph theory, it is often useful to know the distance separating two vertices. In non-weighted graphs, it can be defined as the minimal number of nodes separating these two vertices.

Definition 2.11 (Vertices Distance)**.** *Let* $G = (V, E, \Sigma)$ *be a graph.* The distance between two vertices $v, v' \in V$ *is defined as*

$$\|(v, v')\| \stackrel{\text{def}}{=} \min_{\substack{p \in \mathcal{P}_G \\ v, v' \in p}} (\texttt{index}(v') - \texttt{index}(v))$$

Remark 2.4. *It is worth noting that, in a directed graph* $G = (V, E, \Sigma)$*, the distance between two vertices* $v, v' \in V$ *is usually not commutative, i.e.,* $\|(v, v')\| \neq \|(v', v)\|$*.*

Since a path p represents an ordering of the vertices of the graph, if a vertex appears after another vertex inside p, then the latter is said to be *reachable* from the former.

Definition 2.12 (Vertex Reachability)**.** *Let* $G = (V, E, \Sigma)$ *be a graph, and* $\mathcal{P}_G$ *be its corresponding set of paths. A vertex* $v' \in V$ *is said to be* reachable from another vertex $v \in V$ *if and only if there exists* $p \in \mathcal{P}_G(v)$ *such that* $v' \in p$*. This relationship is written* $v \rightarrow^* v'$*.*

Another useful notion in graph theory concerns the *connectivity* of the vertices.

Definition 2.13 (Connected Graph)**.** *Let* $G = (V, E, \Sigma)$ *be a graph.* G *is said to be* connected *if and only if for all* $v, v' \in V$*, there exists* $p \in \mathcal{P}_G$ *such that* $v \in p \ \wedge \ v' \in p$*.*

Based on this notion, every graph can be divided into one or several *components* representing each a connected subgraph of the original graph that is not part of any larger connected subgraph.

Definition 2.14 (Component)**.** *Let* $G = (V, E, \Sigma)$ *be a graph. A* component *of* G *is a subgraph* $G_S = (V_S, E_S, \Sigma_S) \subseteq G$ *such that:*

- G_S *is connected;*
- *There does not exist* $G'_S = (V'_S \subseteq V, E'_S \subseteq E, \Sigma'_S \subseteq \Sigma)$ *such that* $(V_S \subset V'_S \ \vee \ E_S \subset E'_S) \ \wedge \ G'_S$ *is connected.*

When a component ensures a reachability property between each of its vertices, it is said to be *strongly connected*.

Definition 2.15 (Strongly Connected Component)**.** *Let* $G = (V, E, \Sigma)$ *be a graph. A* strongly connected component *of* G *is a component* $G_S = (V_S, E_S, \Sigma_S)$ *of* G *such that* $\forall v_S, v'_S \in V_S, \ v_S \rightarrow^* v'_S$*.*

To conclude, let us define several useful *graph operators* that will be used in the rest of this thesis.

Definition 2.16 (Graph Operators)**.** *Let* $G = (V, E, \Sigma)$ *be a graph.*

— $G \restriction_{\{v_1,...,v_n\}} \stackrel{\text{def}}{=} (V^\restriction, E^\restriction, \Sigma^\restriction)$ *where*

- $V^\restriction = \{v_1, ..., v_n\} \subseteq V$
- $E^\restriction = \{v \to v' \in E \mid v, v' \in V^\restriction\}$
- $\Sigma^\restriction = \{l \in \Sigma \mid \exists v^\restriction \in V^\restriction \ s.t. \ \sigma(v^\restriction) = l\}$

is the restriction of G *to the subset* $\{v_1, ..., v_n\}$ *of its vertices;*

— $\forall v \in V$, $\texttt{childs}(v) \stackrel{\text{def}}{=} \{v' \in V \mid v \to v' \in E\}$ *is the* children *operator, which returns the set of children nodes of any node* v*;*

— $\forall v \in V$, $\texttt{parents}(v) \stackrel{\text{def}}{=} \{v' \in V \mid v' \to v \in E\}$ *is the* parents *operator, which returns the set of parent nodes of any node* v*;*

— $\forall v \in V$, $\texttt{succ}(v) \stackrel{\text{def}}{=} \{v' \in V \mid v \to^* v'\}$ *is the* succ *operator, which returns the set of successor nodes of any node* v*;*

— $\forall v \in V$, $\texttt{pred}(v) \stackrel{\text{def}}{=} \{v' \in V \mid v' \to^* v\}$ *is the* pred *operator, which returns the set of predecessor nodes of any node* v*;*

— $\forall p \in \mathcal{P}_G$,

$$p[v_s :] = \begin{cases} (v_s, ..., v_n) & \text{if } v_s \in p \\ p & \text{otherwise} \end{cases};$$

$$p[: v_e] = \begin{cases} (v_1, ..., v_e) & \text{if } v_e \in p \\ p & \text{otherwise} \end{cases};$$

$$p[v_s : v_e] = p[v_s :][: v_e]$$

is the slice operator *applied to paths, in its three possible forms.*

2.3 Tree

In Chapter 3, some elements are represented as more constrained graph structures, called *trees*, useful for their numerous properties.

Definition 2.17 (Tree)**.** *A* tree *is an undirected, connected, and acyclic graph defined as a 2-tuple* $T \stackrel{\text{def}}{=} (V, E)$*, where:*

— V *is a set of* vertices*;*

— $E \subseteq V \times V$ *is a set of* edges *connecting the vertices, whose elements are written* $e = v—v'$*.*

In this thesis, we focus in particular on a precise type of tree, which has the properties of being *rooted* and *directed* in a *top-bottom fashion.* Such particular trees are called *out-trees* or *arborescences.*

Definition 2.18 (Out-Tree/Arborescence)**.** *An* out-tree *or* arborescence *is defined as a 3-tuple* $T_O \stackrel{\text{def}}{=} (V, E, R)$*, where:*

- V *is a set of* vertices*;*
- $E \subseteq V \times V$ *is a set of* directed edges *connecting the vertices, whose elements are written* $e = v \to v'$*;*
- $R \in V$ *is the* root vertex*, the only vertex having no incoming transition, i.e.,* $\exists! R \in V$ *such that* $\forall v \in V,\ \nexists e = v \to R \in E$*.*

Each vertex of an out-tree, except the root vertex, has exactly one parent vertex, i.e., $\forall v \in V \setminus \{R\},\ \exists! v_p \in V$ *such that* $v_p \to v \in E$*.*

By definition, cutting an arborescence at a certain height results in a (smaller) arborescence, called *sub-arborescence*.

Definition 2.19 (Sub-Arborescence)**.** *Let* $T_O = (V, E, R)$ *be an arborescence. For all* $v \in V$*, the 3-tuple* (V', E', v) *where*

- $V' \stackrel{\text{def}}{=} \{v_1, v_2 \in V \mid v_1 \to v_2 \in E'\}$*;*
- $E' \stackrel{\text{def}}{=} \{v_1 \to v_2 \in E \mid \neg(v_1 \to^* v)\}$*;*

is also an arborescence, called sub-arborescence *of* T_O*.*

In their generic form, arborescences do not provide any ordering of their vertices. However, in some cases, and in particular in this work, it may be of interest to order the vertices with regards to the others.

Definition 2.20 (Ordered Arborescence)**.** *A* (totally) ordered arborescence *is an arborescence* $T_O = (V, E, R)$ *such that for all* $v \in V$*, for all* $v', v'' \in \texttt{childs}(v),\ v' \preceq v'' \vee v'' \preceq v'$*. It is denoted* $T_O^{\preceq}$*.*

In this thesis, the ordering of the vertices of an ordered arborescence is ensured by an identifier $i \in \mathbb{N}$ attributed to each vertex. By convention, these values are sorted by ascending order, i.e., $\forall (i, j) \in \mathbb{N} \times \mathbb{N}^*,\ v_i \preceq v_{i+j}$. Moreover, the identifiers of the children nodes of any node $v \in V$ must be unique, i.e., $\{v_k, v_l, ..., v_n\} \in \texttt{childs}(v) \Rightarrow k \neq l \neq ... \neq n$. For brevity, as the approaches presented in this thesis only deal with ordered arborescences, the notation $T_{\preceq}$ will be used as simplification of $T_O^{\preceq}$ to represent ordered arborescences. To conclude, we define some *arborescences operators* useful for the rest of this thesis.

Definition 2.21 (Arborescences Operators)**.** *Let* $T_{\preceq} = (V, E, R)$ *be an ordered arborescence. We define the following operators on* $T_{\preceq}$*:*

- $\forall v \in V,\ T_{\preceq}(v)$ *returns the* sub-arborescence $T'_{\preceq} = (V', E', v)$ *rooted by* v*;*
- $\texttt{root}(T_{\preceq}) \stackrel{\text{def}}{=} R$ *returns the* root *of the arborescence;*
- $\forall v \in V \setminus \{R\},\ \texttt{parent}(v) = v'$ *such that* $v' \to v \in E$ *returns the* parent node *of* v*;*
- $\forall v \in V, \forall i \in [0...|\texttt{childs}(v)| - 1], \texttt{childs}(v)[i]$ *returns the* i^{th} child *of* v*.*[5]

[5]Note that this operator is only definable because we consider ordered arborescences.

2.4 BPMN

2.4.1 Syntax & Representation

Syntax

BPMN 2.0 (BPMN, as a shorthand, in the rest of this manuscript) was published as an ISO/IEC standard in 2013 and is nowadays extensively used for modelling and developing business processes. This thesis focuses on activity diagrams including the BPMN constructs related to control-flow modelling and behavioural aspects.

Specifically, the node types *event*, *task*, and *gateway*, and the edge type *sequence flow* are considered. Start and end events are used, respectively, to initialise and terminate processes. While there must be a unique start event in a BPMN process, there might be several end ones. A task represents an atomic activity that has (or can be rewritten with) exactly one incoming and one outgoing flow. Despite not belonging to the standard definition of BPMN, several works enriched the notation with resources [Gro07, AGMW08, SCV11, BDGP16] and time [GT09, WG09, AERDM16] for tasks. In this thesis, both of these additional quantitative aspects are used, as a mean to increase the realism of the considered processes. Thus, a task may have a duration or delay, expressed by default in units of time (UT). It can also be defined using probabilistic distributions, in case of non-fixed duration. Resource requirements are explicitly defined at the task level. A task can include, as part of its specification, the resources it requires to be executed. In such a case, it means that the task needs the specified number of replicas or instances of such resources to be able to start. Once the resources are acquired, the task executes for the specified duration. The acquisition of a resource is achieved in a first-come-first-served strategy. If a task needs more resource replicas than available, it remains in a waiting state until the release of a sufficient number of replicas of the required resources. When the execution of the task is completed, the resources it acquired for its execution are released.

A sequence flow connects two nodes executed one after the other in a specific execution order. Gateways are used to control the divergence and the convergence of the execution flow. In this work, the two main kinds of gateways used in activity diagrams are considered, namely, *exclusive* and *parallel* gateways. Gateways with one incoming flow and multiple outgoing flows are called *splits*, e.g., split parallel gateway, while gateways with one outgoing flow and multiple incoming flows are called *merges*, e.g., merge parallel gateway. A parallel gateway creates concurrent executions for all its outgoing flows or synchronises concurrent executions of all its incoming flows. An exclusive gateway chooses one out of a set of mutually exclusive alternative incoming or outgoing flows. As only one branch of an exclusive gateway is executed, a selection has to be done among the available branches. In this work, this selection is based on the probability of execution of the branches, which are represented as a real value ranging between 0 and 1. These probabilities must sum to 1, and are either given directly by the designer of the process, inferred from execution traces (using, for instance, process mining techniques [vdA16]), or considered as equal for each

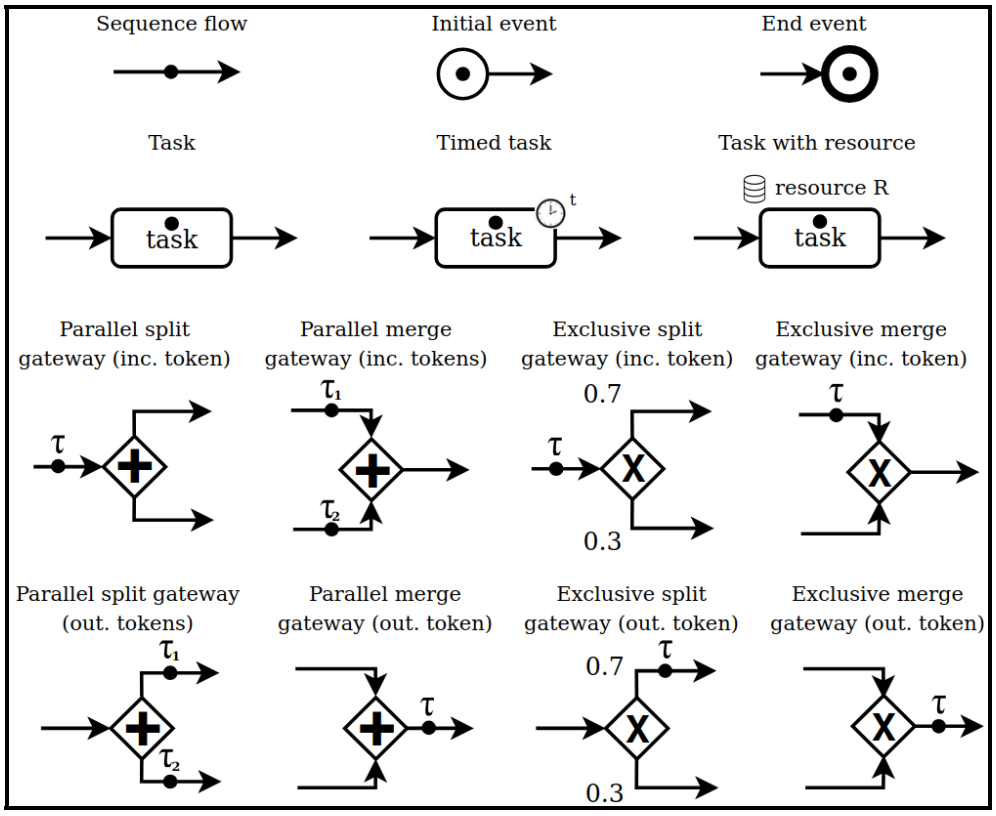


Figure 2.1: Supported Fragment of the Extended BPMN syntax

branch of the gateway otherwise. Such gateways can also be used to represent repetitive behaviours (i.e., loops). This excerpt of the BPMN syntax, sufficient to cover more than 90% of the BPMN processes in practice [KPS19], is given in Figure 2.1. The following notations will be used in the rest of this thesis:

— t represents a *task* of label t;
— s represents a *start event*;
— e represents an *end event*;
— $+_S$ represents an *exclusive split gateway*;
— $+_M$ represents an *exclusive merge gateway*;
— $\times_S$ represents a *parallel split gateway*;
— $\times_M$ represents a *parallel merge gateway*.

It is worth noting that these notations are almost identical to the ones presented in Figure 2.1, with the particularity of being labelled with the first letter of their name (e.g., *e* for the end event) to ease their remembering, and that gateways are differentiated both on their type (i.e., *parallel* or *exclusive*) and their role (i.e., *split 'S'* or *merge 'M'*).

Representation

Due to its structure, it is suitable to represent a BPMN process as a graph $G = (V, E, \Sigma)$, where a vertex $v \in V$ is either an initial event, an end event, a task, or a gateway (exclusive or parallel, split or merge), and an edge $e \in E$ is a sequence flow. The set of labels Σ corresponds to the labels (or names) of the tasks of the process. To stick to the usual representation of BPMN processes, vertices will be named *nodes* while edges will be named *(sequence) flows* in the rest of this manuscript. We now define several *BPMN operators* which will be helpful in the rest of this thesis.

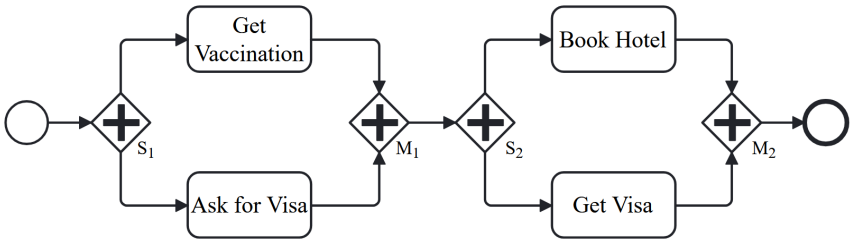
Definition 2.22 (BPMN Operators)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process.*

- *For all* $v \in V$, $\theta(v) \in \{$  $,$  $,$  $,$  $,$  $,$  $,$  $\}$ *returns the* type *of the given vertex;*
- *For all* $v \in V$ *such that* $\theta(v) =$  $,\ \delta(v) = d \in \mathbb{N}$ *returns the* duration d *of the task* v *expressed in units of time (UT);*
- $\mathcal{P}_G^0 \stackrel{\text{def}}{=} \{p = (v_1, ..., v_n) \in \mathcal{P}_G \mid \theta(v_1) =$  $\wedge\ \theta(v_n) =$  $\}$ *is the* set of paths *of* G *starting from an initial event and ending with an end event;*
- *For all* $p \in \mathcal{P}_G^0$, $\texttt{tasks}(p) \stackrel{\text{def}}{=} \{v \in p \mid \theta(v) =$  $\}$ *returns the* set of tasks *of the given path.*

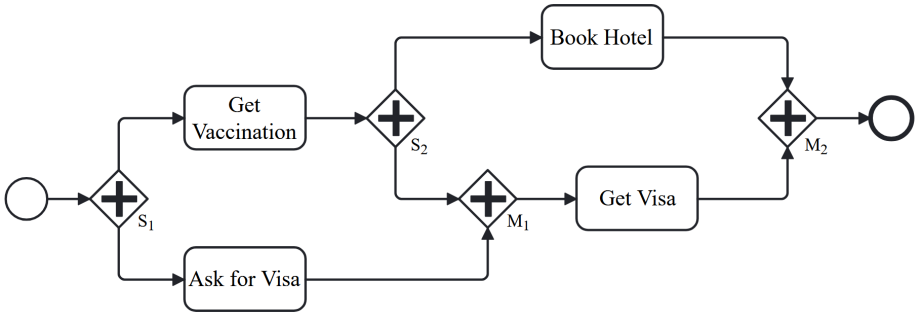
A BPMN process is said to be *balanced* when each of its split gateways (independently of its type) has a 1-to-1 mapping with a merge gateway of the same type. This implies that each branch initiated with a split gateway is explicitly synchronised at a single convergence point. Such balancing decomposes the process in tasks and blocks consisting of (balanced) parallel structures, (balanced) choice structures, and (balanced) loops. Balanced parallel (resp. choice) structures typically start with a parallel (resp. exclusive) split gateway followed by several branches, and terminate with a parallel merge (resp. exclusive) gateway. Balanced loop structures are usually composed of two parts: the *body*—which corresponds to the part of the loop that is necessarily executed—starts with an exclusive merge gateway and terminates with an exclusive split gateway, and the *optional part*—which corresponds to the part of the loop that may be skipped—starts with the exclusive split gateway that ends the body, and terminates with the exclusive merge gateway that starts the body.

However, the BPMN standard does not require processes to be balanced. Indeed, in practice, it is rather common to find processes that are unbalanced, containing for instance partial gateway closure, or intricated loops. The approach presented in Chapter 3 deals with both kinds of processes, while the ones detailed in Chapter 4 are for now restricted to balanced processes.

Example. Let us illustrate this notion of balancing/unbalancing on a very simple BPMN process, shown in Figure 2.2. The top process (Figure 2.2(a)) is balanced. Indeed, each parallel split gateway has a corresponding merge gateway, and oppositely. On the other hand, the bottom process (Figure 2.2(b)) is unbalanced. As the reader can see, the paral-



(a) Example of Balanced BPMN Process



(b) Example of Unbalanced BPMN process

Figure 2.2: Example of Balanced and Unbalanced BPMN Processes

lel split gateway S_2 is partially closed by the parallel merge gateway M_1, and is thus not uniquely related to the parallel merge gateway M_2. Such constructions, although being more complex than their siblings, are particularly useful to represent partial synchronisation. For instance, here, the task Book Hotel could be independent of the task Ask for Visa. However, in the balanced version of the process, it can not be performed before it. Here, as seen in the second figure, the unbalancing allows this behaviour, as task Book Hotel now only depends on the completion of task Get Vaccination to be executed.

Minimisation

Depending on its structure, a BPMN process may not be *minimal*. This notion of minimality is strongly correlated to the fact that several syntactically different BPMN processes may depict exactly the same behaviour, i.e., be semantically equivalent according to strong bisimulation [Par81]. Among them, the minimal one is the one containing the fewest number of nodes, or, in case of equality, the fewer number of edges.

Definition 2.23 (Minimal BPMN Process)**.** *Let* $\{G_1 = (V_1, E_1, \Sigma_1), ..., G_n = (V_n, E_n, \Sigma_n)\}$ *be a set of semantically equivalent yet syntactically different BPMN processes. The* minimal BPMN process *of this set is the graph* $G_{min} = (V_{min}, E_{min}, \Sigma_{min})$ *such that* $\forall i \in [1...n],\ (|V_i| \geq |V_{min}|)\ \wedge\ (|V_i| = |V_{min}| \Rightarrow |E_i| > |E_{min}|)$.

This minimal BPMN process can be obtained by applying repeatedly a set of *minimisation rules*, until reaching a fixed point.

Definition 2.24 (Minimisation Rules)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process. G can be minimised by applying* a set of minimisation rules *on its gateways. This minimisation operation, repeated as many times as needed to reach a fixed point, is defined as*

$$\texttt{min}(G) \stackrel{\text{def}}{=} \bigcirc_{i \in [1...5]} \texttt{min}_i(G)$$

where

— $\texttt{min}_1(G) \stackrel{\text{def}}{=} \begin{cases} (V \setminus \{v'\}, E', \Sigma) & \text{if } \exists v \to v' \in E \mid \theta(v) = \theta(v') \in \{\langle\!\times\!\rangle_S, \langle\!+\!\rangle_S\} \\ G & \text{otherwise} \end{cases}$

with $E' = E \setminus \{v \to v'\} \setminus \bigcup_{v_c \in \texttt{childs}(v')} \{v' \to v_c\} \cup \bigcup_{v_c \in \texttt{childs}(v')} \{v \to v_c\}$,

— $\texttt{min}_2(G) \stackrel{\text{def}}{=} \begin{cases} (V \setminus \{v\}, E', \Sigma) & \text{if } \exists v \to v' \in E \mid \theta(v) = \theta(v') \in \{\langle\!\times\!\rangle_M, \langle\!+\!\rangle_M\} \\ G & \text{otherwise} \end{cases}$

with $E' = E \setminus \{v \to v'\} \setminus \bigcup_{v_p \in \texttt{parents}(v)} \{v_p \to v\} \cup \bigcup_{v_p \in \texttt{parents}(v)} \{v_p \to v'\}$,

— $\texttt{min}_3(G) \stackrel{\text{def}}{=} \begin{cases} (V, E \setminus \{v \to v'\}, \Sigma) & \text{if } \exists v \to v' \in E \mid \theta(v) = \langle\!+\!\rangle_S \ \wedge \ \theta(v') = \langle\!+\!\rangle_M \\ G & \text{otherwise} \end{cases}$

— $\texttt{min}_4(G) \stackrel{\text{def}}{=} \begin{cases} (V \setminus \{v\}, E', \Sigma) & \text{if } \exists v \in V \mid \theta(v) \in \{\langle\!\times\!\rangle_S, \langle\!+\!\rangle_S\} \ \wedge \ |\texttt{childs}(v)| = 1 \\ G & \text{otherwise} \end{cases}$

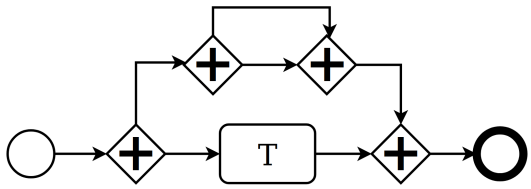
with $E' = E \setminus \{v_p = \texttt{any}(\texttt{parents}(v)) \to v, v \to v_c = \texttt{any}(\texttt{childs}(v))\} \cup \{v_p \to v_c\}$,

— $\texttt{min}_5(G) \stackrel{\text{def}}{=} \begin{cases} (V \setminus \{v\}, E', \Sigma) & \text{if } \exists v \in V \mid \theta(v) \in \{\langle\!\times\!\rangle_M, \langle\!+\!\rangle_M\} \ \wedge \ |\texttt{parents}(v)| = 1 \\ G & \text{otherwise} \end{cases}$

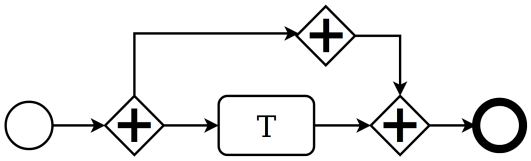
with $E' = E \setminus \{v_p = \texttt{any}(\texttt{parents}(v)) \to v, v \to v_c = \texttt{any}(\texttt{childs}(v))\} \cup \{v_p \to v_c\}$.

These rules can be represented graphically, as shown in Figure 2.3. Figure 2.3(b), representing rule $\texttt{min}_1$, depicts the fusion of two consecutive split gateways of same type. Figure 2.3(d), representing rule $\texttt{min}_2$, illustrates the fusion of two consecutive merge gateways of same type. Figure 2.3(f), representing rule $\texttt{min}_3$, describes the removal of a useless (empty) flow of a parallel gateway. Figure 2.3(h) (resp. 2.3(j)), representing rule $\texttt{min}_4$ (resp. $\texttt{min}_5$), shows the removal of a useless parallel split (resp. merge) gateway.

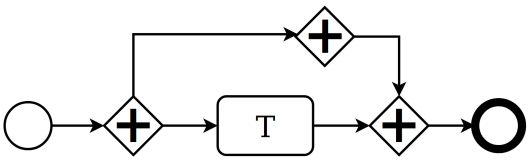
Remark 2.5. *These rules are complete and deterministic, in the sense that there exists no other minimisation rule that can be applied to the subset of BPMN handled in this thesis, and that, disregarding their order of application, these rules will always lead to the same final unique minimal BPMN process. For instance, enlarging the scope of rule* $\texttt{min}_3$ *to handle exclusive gateways would modify the semantics of the original process, as an exclusive gateway with an empty flow has a well-defined semantics in BPMN.*



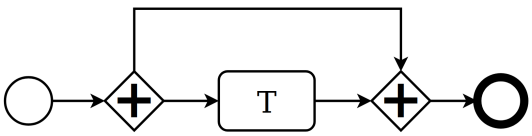
(a) BPMN Process Before Applying the Minimisation Rule $\texttt{min}_1$



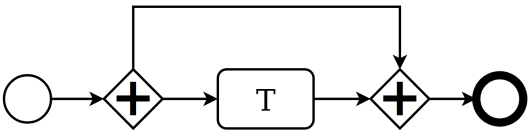
(b) BPMN Process After Applying the Minimisation Rule $\texttt{min}_1$



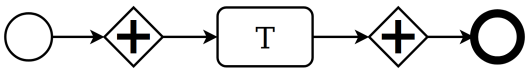
(c) BPMN Process Before Applying the Minimisation Rule $\texttt{min}_2$



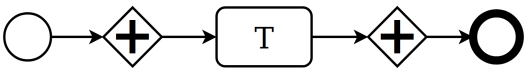
(d) BPMN Process After Applying the Minimisation Rule $\texttt{min}_2$



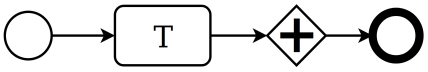
(e) BPMN Process Before Applying the Minimisation Rule $\texttt{min}_3$



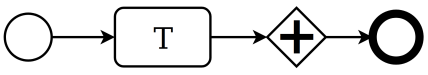
(f) BPMN Process After Applying the Minimisation Rule $\texttt{min}_3$



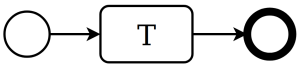
(g) BPMN Process Before Applying the Minimisation Rule $\texttt{min}_4$



(h) BPMN Process After Applying the Minimisation Rule $\texttt{min}_4$



(i) BPMN Process Before Applying the Minimisation Rule $\texttt{min}_5$



(j) BPMN Process After Applying the Minimisation Rule $\texttt{min}_5$

Figure 2.3: Illustration of the Minimisation Rules

2.4.2 Resources

In this work, tasks may require resources to be executed. This representation models the fact that, in practical situations, a task will probably require some resources to be executed, such as an employee or a machine. For instance, to perform a task such as packaging a good, one may require a worker to put the good in the box, and a tape machine to tape the box. However, the model does not yet handle consumable resources, and considers that a resource can be used infinitely often, as long as it is available.

In the rest of this thesis, a resource will be represented with the symbol r, and will have a label (or name) l, retrievable with the operator $\sigma(r)$. The set of resources required by the tasks of a BPMN process to execute will be denoted S_{res}. Similarly to graphs, the set of all resource names of a BPMN process (i.e., the resources in S_{res}) will be denoted Σ_{res}. The resources required by a given task of a BPMN process can be retrieved using a *task resource operator*.

Definition 2.25 (Task Resource Operator)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process. For all $v \in V$ such that $\theta(v) =$*  *, we define the* task resource operator $\rho : V \to (S_{res} \to \mathbb{N})$ *as*

$$\rho(v)(r) \stackrel{\text{def}}{=} n$$

where n is the number of replicas of r required by v, or 0 if v does not require any replica of r to execute.

In a real-world context, a resource is usually associated to a cost that describes the amount of money required to use this resource for a given duration. Without loss of generality, we consider that each task r is associated to a cost c given in dollars per unit of time ($/UT). The cost of a resource can be retrieved using the operator $\texttt{cost}(r)$. Similarly, the number of resources usually available is finite. To stick to this representation, we make use of a *resource pool* to map every resource of S_{res} to its available number of replicas.

Definition 2.26 (Resource Pool)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process and let S_{res} be the set of resources required by G's tasks to execute. A* resource pool *of G, noted P, is a set of 2-tuples consisting of a resource of S_{res} and its number of available replicas, defined as*

$$P \stackrel{\text{def}}{=} \{(r, n) \in S_{res} \times \mathbb{N} \mid n \text{ is the number of available replicas of } r\}$$

Remark 2.6. *It is worth noting that the (simplified) resources model adopted in this thesis considers that resources are available continuously, e.g., an employee never takes a coffee/lunch break.*

For a BPMN process to be executable, the available resource pool of the process must contain at least the maximum number of replicas of each resource required by a task of the process. This pool of resources is called *minimal resource pool*.

Definition 2.27 (Minimal Resource Pool)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process*

and let S_{res} be the set of resources required by G's tasks to execute. The minimal pool of resources *of G is defined as*

$$P_{min} = \{(r, n) \in S_{res} \times \mathbb{N} \mid n = \max_{\substack{v \in V \\ \theta(v) = \boxed{t}}} \rho(v, r)\}$$

To facilitate the usage and the readability of resource pools, we introduce several useful *operators*.

Definition 2.28 (Resource Pool Operators)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process and let P be a resource pool of G. We define the following* operators *on P:*

- $\forall (r, n) \in P,\ P(r) = n$ returns the number of available replicas *of r in P;*
- $\forall (r, n) \in P,\ \forall n' \in \mathbb{N},\ P[r \mapsto n'] = P'$ updates the number of available replicas *of r to n' in P such that $P[r \mapsto n'](r) = n'$.*

Finally, we provide a comprehensive definition of the *comparability* of two resource pools.

Definition 2.29 (Resource Pools Comparison)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process and let P, P' be two resource pools of G. We define the following* comparisons *of P and P':*

- $P \subseteq P' \Leftrightarrow \forall (r, n) \in P,\ P(r) \leq P'(r)$;
- $P \subset P' \Leftrightarrow \forall (r, n) \in P,\ P(r) < P'(r)$;
- $P \nsubseteq P' \Leftrightarrow \neg(P \subseteq P')$;
- $P \not\subset P' \Leftrightarrow \neg(P \subset P')$;
- $P = P' \Leftrightarrow (P \subseteq P') \wedge (P' \subseteq P)$;
- $P \neq P' \Leftrightarrow \neg(P = P')$.

2.4.3 Execution Flow

In BPMN, the execution flow of a process is controlled by tokens, represented with the $\bullet$ symbol and denoted τ (as shown in Figure 2.1). These tokens circulate through the process from its start event to its end events, and can be produced and/or consumed depending on the control-flow elements that they encounter. The position of the tokens at a given time indicates which portions of the process are currently being executed.

Both nodes and flows can hold token. A node holding a token is written $\dot{v}$, while a flow $e = v \to v' \in E$ holding a token is written either $v \xrightarrow{\bullet} v'$ or $\dot{e}$. To complete its execution, a node either transmits its token to (one of) its outgoing flow(s), consumes its incoming tokens and produces a new one, or produces several tokens, depending on its type. It is worth noting that processes are not necessarily *safe*, i.e., a node may hold several tokens at the same time. A task must acquire the resources that it needs to execute (if any) and wait for its duration (if any) before being allowed to transmit its token to its (unique)

outgoing flow. An exclusive split gateway non-deterministically sends its token to one of its outgoing flows only, while an exclusive merge gateway transmits the token that it received to its (unique) outgoing flow. A parallel split gateway produces new tokens and sends one to each of its outgoing flows, while a parallel merge gateway consumes the tokens that it received from its incoming flows, and produces a new token that is sent to its (unique) outgoing flow. It is worth noting that a parallel merge gateway must wait for each of its incoming flows to hold a token before consuming these tokens and produce a new one sent to its outgoing flow.

The current state of a BPMN process, i.e., the number of tokens currently hold by the nodes/edges of that process, is called a *configuration*.

Definition 2.30 (Configuration)**.** *Let* $G = (V,\ E,\ \Sigma)$ *be a BPMN process. A* configuration *is a set*

$$C \stackrel{\text{def}}{=} \{(v, m) \in V \times \mathbb{N},\ \forall v \in V\} \cup \{(e, n) \in E \times \mathbb{N},\ \forall e \in E\}$$

where m *and* n *represent respectively the number of tokens currently hold by a vertex* v *or an edge* e*.*

To facilitate the usage and the readability of configurations, we introduce several useful *operators*.

Definition 2.31 (Configurations Operators)**.** *Let* $G = (V,\ E,\ \Sigma)$ *be a BPMN process, let* C *be any configuration of* G *and let* $A = V \cup E$*. We define the following* operators *on* C*:*

- $\forall a \in A,\ C(a) = k$ returns the number of tokens *currently hold by* a *in* C*;*
- $\forall a \in A,\ \forall k' \in \mathbb{N},\ C[a \mapsto k'] = C'$ updates the number of tokens *currently hold by* a *in* C *such that* $C[a \mapsto k'](a) = k'$*.*

The *initial configuration* of a BPMN process, called C_i, is a configuration in which only the initial event of the process holds a token.

Definition 2.32 (Initial Configuration)**.** *Let* $G = (V,\ E,\ \Sigma)$ *be a BPMN process. The* initial configuration *of* G *is a configuration* C_i *defined as*

$$C_i \stackrel{\text{def}}{=} \{(v, 0) \mid v \in V \wedge \theta(v) \neq \textcircled{s}\} \cup \{(e, 0),\ \forall e \in E\} \cup \{(v_s, 1) \mid v_s \in V \wedge \theta(v_s) = \textcircled{s}\}$$

In the context of BPMN processes enriched with durations for tasks, there is a need for a *global timer* that is in charge of simulating discrete increases of the time based on a global clock, such that each tick of its clock represents one unit of time (1UT).

Definition 2.33 (Global Timer)**.** *Let* $G = (V,\ E,\ \Sigma)$ *be a BPMN process. The* global timer *of* G *is represented as a set of pairs, each containing a task and its remaining exe-*

cution time. It is defined as

$$\mathfrak{T} \stackrel{\text{def}}{=} \{(v,n) \in V \times \mathbb{N} \mid \theta(v) = \boxed{\ t\ } \ \wedge \ n \in [1...\delta(v)]\}$$

To facilitate the usage and the readability of global timers, we introduce several useful *operators.*

Definition 2.34 (Global Timer Operators)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process and let* $\mathfrak{T}$ *be its global timer. We define the following operators on* $\mathfrak{T}$*:*

- $\forall (v,n) \in \mathfrak{T},\ \mathfrak{T}(v) = n$ returns the remaining execution time *of task* v*;*
- $\forall (v,n) \in \mathfrak{T},\ \forall n' \in [1...\delta(v)],\ \mathfrak{T}[v \mapsto n'] = \mathfrak{T}'$ updates the remaining execution time *of task* v *to* n' *such that* $\mathfrak{T}[v \mapsto n'](v) = n'$*.*

Every 1UT, the global timer makes a tick which makes the process move from one configuration to another. This change of configuration is ruled by the multiple semantic rules shown in Figures 2.4, 2.5, and 2.6.

The sanity rule of Figure 2.4 performs a first check to ensure that only the elements of the process currently holding tokens will be considered for the transition. The sequential composition rule details how the movement of multiple tokens in the process is handled, with a particular attention provided to the resulting configuration and the resulting global timer.

The parallel split rule of Figure 2.5 states that if a token holder is a parallel split gateway, its outgoing flows will possess as many tokens as it possesses in the resulting configuration, while the parallel split will not possess tokens anymore. The exclusive split rule states that if a token holder is an exclusive split gateway, a probabilistically chosen set of its outgoing flows will possess one more token, while the exclusive split will not possess tokens anymore. The finished task rule states that when the global timer contains a task of remaining duration equal to 1, it will send its token to its child in the next configuration, its resources will be released, and its execution information will be removed from the global timer. The non-finished task rule states that when the global timer contains a task of remaining duration greater than 1, the global timer of the next configuration will simply decrease this duration by 1UT. The general rule for vertices states that for all the other vertices, all the tokens hold by these vertices are transmitted to their child in the next configuration, while their number of tokens is set to 0.

The start task rule of Figure 2.6 applies if the outgoing vertex of an edge is a task. For each token hold by the incoming flow of that task, the current pool of resources will be reduced by the number of resources required by the task, the global timer will add the duration of that task as remaining time of the current instance of that task, and the number of tokens hold by that task will be increased by one in the next configuration, while the number of tokens hold by its incoming flow will be reduced by 1. The parallel merge rule verifies the number of tokens hold by the incoming flows of a parallel merge gateways, and if all these

$$\frac{\widehat{A} \subseteq A \mid (\forall i \in [0...n],\ C(a_i) \neq 0) \wedge (\forall a_j \in A \setminus \widehat{A},\ C(a_j) = 0) \quad (C, P, \mathfrak{T}) \xrightarrow{\widehat{A}} (C_A, P_A, \mathfrak{T}_A)}{(C, P, \mathfrak{T}) \xrightarrow{A} (C_A, P_A, \mathfrak{T}_A)}$$

where $\widehat{A} = \{a_0, ..., a_n\}$.

(a) Sanity Rule

$$\frac{(C, P, \mathfrak{T}) \xrightarrow{\{a_0\}} (C_{a_0}, P_{a_0}, \mathfrak{T}_{a_0}) \qquad (C, P_{a_0}, \mathfrak{T}) \xrightarrow{\{a_1,...,a_n\}} (C_{a_n}, P_{a_n}, \mathfrak{T}_{a_n})}{(C, P, \mathfrak{T}) \xrightarrow{\{a_0,a_1,...,a_n\}} (C_{a_0} \oplus C_{a_n}, P_{a_n}, \mathfrak{T}_{a_0} \oplus \mathfrak{T}_{a_n})}$$

where:

- $C_{a_0} \oplus C_{a_n} = [a_0 \mapsto x_0; ...; a_n \mapsto x_n]$ *with,* $\forall i \in [0...n],\ x_i = C_{a_0}(a_i) + C_{a_n}(a_i) - C(a_i)$;
- $\mathfrak{T}_{a_0} \oplus \mathfrak{T}_{a_n} = \bigcup_{((a_0,x_0),(a_n,x_n)) \in \mathfrak{T}_{a_0} \times \mathfrak{T}_{a_n}} \{(a_0, \min\{x_0, x_n\}) \mid a_0 = a_n\}$.

(b) Sequential Composition Rule

Figure 2.4: General Transition Rules

flows hold at least one token, the minimum number of tokens hold by these incoming flows is removed from them and added to the parallel merge gateway in the next configuration. The general rule for edges simply transmits the tokens hold by an edge to the outgoing vertex of that edge in the next configuration.

The application of these rules is called a *configuration transition.*

Definition 2.35 (Configuration Transition)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process, let* C *be any configuration of* G*, let* P *be any resource pool of* G *such that* $P_{min} \subseteq P$*, and let* $\mathfrak{T}$ *be the global timer of* G*. Every tick of the global timer triggers a* configuration transition *which pushes forward all the tokens of the current configuration once, according to the transition rules presented in Figures 2.4, 2.5, and 2.6. This is written*

$$\texttt{push}((C, P, \mathfrak{T})) = (C', P', \mathfrak{T}')$$

where C' *is the resulting configuration,* P' *the resulting pool of resources, and* $\mathfrak{T}'$ *the resulting global timer.*

The execution of a sound BPMN process terminates in a configuration in which only end events hold tokens, called *final configuration.*

Definition 2.36 (Final Configuration)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process, let* P *be any resource pool of* G *such that* $P_{min} \subseteq P$*, and let* $\mathfrak{T}$ *be the global timer of* G*. A* final

$$\frac{(a,n) \in C \qquad \theta(a) = \diamondsuit_S \qquad \forall a' \in \texttt{childs}(a)}{(C, P, \mathfrak{T}) \xrightarrow{\{a\}} (C[(a,a') \mapsto C((a,a')) + n; a \mapsto 0], P, \mathfrak{T})}$$

(a) Parallel Split Rule

$$\frac{(a,n) \in C \qquad \theta(a) = \diamondsuit_S \qquad \forall a' \in \texttt{any}_p(\texttt{childs}(a), n)}{(C, P, \mathfrak{T}) \xrightarrow{\{a\}} (C[(a,a') \mapsto C((a,a')) + 1; a \mapsto 0], P, \mathfrak{T})}$$

(b) Exclusive Split Rule

$$\frac{(a,n) \in C \qquad \theta(a) = \boxed{t} \qquad \mathfrak{T}(a) = 1 \qquad \forall r \in \rho(a) \qquad a' = \texttt{childs}(a)[0]}{(C, P, \mathfrak{T}) \xrightarrow{\{a\}} (C[(a,a') \mapsto C((a,a')) + n; a \mapsto 0], P[r \mapsto P(r) + \rho(a)(r)], \mathfrak{T} \setminus \{(a,1)\})}$$

(c) Finished Task Rule

$$\frac{(a,n) \in C \qquad \theta(a) = \boxed{t} \qquad \mathfrak{T}(a) > 1}{(C, P, \mathfrak{T}) \xrightarrow{\{a\}} (C, P, \mathfrak{T}[a \mapsto \mathfrak{T}(a) - 1])}$$

(d) Non Finished Task Rule

$$\frac{(a,n) \in C \qquad \theta(a) \notin \{\diamondsuit_S, \diamondsuit_S, \boxed{t}\} \qquad a' = \texttt{childs}(a)[0]}{(C, P, \mathfrak{T}) \xrightarrow{\{a\}} (C[(a,a') \mapsto C((a,a')) + n; a \mapsto 0], P, \mathfrak{T})}$$

(e) General Rule for Vertices

Figure 2.5: Vertices Transition Rules

$$\frac{((v,v'),n) \in C \qquad \theta(v') = \boxed{t} \qquad \forall i \in [1...n](\forall r \in \rho(v'),\ \rho(v')(r) \leq P(r))}{(C,P,\mathfrak{T}) \xrightarrow{\{a\}} (C[v' \mapsto C(v')+1; a \mapsto n-1], P[r \mapsto P(r)-\rho(v')(r)], \mathfrak{T}[v'_i \mapsto \delta(v')])}$$

(a) Start Task Rule

$$\frac{((v,v'),n) \in C \qquad \theta(v') = \diamondsuit_M \qquad m = \min_{v_p \in V_p}\{C((v_p,v'))\} \qquad V_p = \{v_p^0,...,v_p^s\} = \texttt{parents}(v')}{(C,P,\mathfrak{T}) \xrightarrow{\{a\}} (C[v' \mapsto C(v')+m; v_p^0 \mapsto C(v_p^0)-m; ...; v_p^s \mapsto C(v_p^s)-m], P, \mathfrak{T})}$$

(b) Parallel Merge Rule

$$\frac{((v,v'),n) \in C \qquad \theta(v') \notin \{\diamondsuit_M, \boxed{t}\}}{(C,P,\mathfrak{T}) \xrightarrow{\{a\}} (C[v' \mapsto C(v')+n; a \mapsto 0], P, \mathfrak{T})}$$

(c) General Rule for Edges

Figure 2.6: Edges Transition Rules

configuration *of* G *is a configuration* C_f *defined as*

$$C_f \stackrel{\text{def}}{=} \{(v,0),\ \forall v \in V \setminus V_{end}\} \ \cup\ \{(e,0),\ \forall e \in E\}$$

where $V_{end} = \{v \in V \mid \theta(v) = \textcircled{e}\}$.

Remark 2.7. *By definition, a final configuration is idempotent regarding the* `push` *operation, i.e.,* $\texttt{push}((C_f,P,\mathfrak{T})) = (C_f,P,\mathfrak{T})$ *for any final configuration* C_f.

The list of configurations that a process went through during its execution is stored in a *history*.

Definition 2.37 (History)**.** *Let* $G = (V,\ E,\ \Sigma)$ *be a BPMN process. An* history *of* G*, noted* $\mathcal{H}$*, is a n-tuple* $((C_1,P_1,\mathfrak{T}_1)...,(C_n,P_n,\mathfrak{T}_n))$*, such that:*

- $\forall i \in [1...n]$:
  - C_i *is a* configuration*;*
  - P_i *is a* resource pool*;*
  - $\mathfrak{T}_i$ *is a* global timer*;*
- C_1 *is an* initial configuration*;*
- $\forall i \in [1...n-1],\ (C_{i+1},P_{i+1},\mathfrak{T}_{i+1}) = \texttt{push}((C_i,P_i,\mathfrak{T}_i))$.

It can be retrieved using the operator $\mathcal{H}(G)$.

If a BPMN process is sound, it necessarily terminates, thus its history is said to be *complete*.

Definition 2.38 (Complete History)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process. A* complete history *of G is an history $\mathcal{H}_C = ((C_1, P_1, \mathfrak{T}_1)..., (C_n, P_n, \mathfrak{T}_n))$ such that C_n is a final configuration.*

The execution of a process can also be defined in terms of *traces*, each of which representing a possible ordering of the tasks of the process, ruled by its possible execution.

Definition 2.39 ((Execution) Trace)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process and let $S_{\mathcal{H}}$ be the set of all possible histories of G. A* trace *of G is a tuple $\lambda = (v_1, ..., v_n)$ such that:*

- $\forall i \in [1...n],\ \theta(v_i) = \boxed{t}$;
- *There exists $\mathcal{H} \in S_{\mathcal{H}}$ such that the $(v_1, ..., v_n)$ appear in the exact same order in λ and $\mathcal{H}$.*

The set of all traces *of a process is written Λ, and can be retrieved using the operator $\Lambda(G)$.*

Remark 2.8. *The notions presented in this section suppose that the considered BPMN process is enriched with both durations and resources. However, they remain valid in a simpler context where the BPMN process does not have durations nor resources. The difference lies in the usage of a resource pool P and a global timer $\mathfrak{T}$, that are no longer useful in the simple context. This affects the transition rules presented in Figures 2.4, 2.5, and 2.6 with the small variations highlighted in red. It is also worth noticing that, if resources and durations are not considered, rule* (5) *and* (6) *are strictly equivalent to rule* (7)*, and rule* (8) *to rule* (9)*. Thus, rules* (5)*,* (6)*, and* (8) *are discarded in this case.*

2.4.4 Metrics

Several performance indicators or metrics can be computed on a business process extended with quantitative aspects, such as execution times, synchronisation/waiting times, resource usage, or total costs. The *execution time* of a process represents the difference between the time at which a token reaches an end event and the time at which the initial token was sent away from the initial event of that process. In our discrete context, it can be seen as the number of clock ticks required to reach a final configuration from an initial one. In other words, it is the number of elements belonging to a complete history of the process.

Definition 2.40 (Execution Time)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process, let P be a pool of resources, and let $\mathcal{H}_C$ be a complete history of G obtained by executing it with resource pool P. The* execution time *of G with regards to $\mathcal{H}_C$ can be defined as*

$$ET_{\mathcal{H}_C}(G, P) \stackrel{\text{def}}{=} |\mathcal{H}_C|$$

In case of conditional structures, the execution of a process may vary. So does its history. However, there is one interesting notion for reasoning on quantitative aspects of a process containing such structures, that is, its *worst-case execution time.*

Definition 2.41 (Worst-Case Execution Time)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process, let P be a pool of resources, and let $S_{\mathcal{H}_C}$ be the set of complete histories of G, representing each of its possible executions with resource pool P. The* worst-case execution time *of G corresponds to the time taken by the longest execution of G to complete, that is*

$$WCET(G, P) \stackrel{\text{def}}{=} \max_{\mathcal{H}_C \in S_{\mathcal{H}_C}} ET_{\mathcal{H}_C}(G, P)$$

However, the worst-case execution time of a process is not necessarily computable. Indeed, a process containing loops has an infinite number of complete histories, as the probability of doing one more iteration of the loop can never reach 0. To palliate this issue, we provide a slightly different version of the worst-case execution time, based on a finite set of complete histories. This notion is called *stochastic worst-case execution time.*

Definition 2.42 (Stochastic Worst-Case Execution Time)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process, let P be a pool of resources, and let $S^{0.01}_{\mathcal{H}_C} = \{\mathcal{H}^1_C, ..., \mathcal{H}^n_C\}$ be the set of complete histories of G, representing each of its possible executions in which loops probabilities do not exceed the threshold of 0.01 (i.e., 1%) with resource pool P. The* stochastic worst-case execution time *of G corresponds to the time taken by the longest such execution of G to complete, that is*

$$SWCET(G, P) \stackrel{\text{def}}{=} \max_{i \in [1...n]} ET_{\mathcal{H}^i_C}(G, P)$$

As the set of complete histories representing the possible executions of a process in which loops probabilities do not exceed the threshold of 0.01 is necessarily defined, the stochastic worst-case execution time of a process is always properly defined. Given a process execution, one can also obtain the *average resources usage* of the process, which corresponds to the resources usage of the process aggregated over time.

Definition 2.43 (Average Resources Usage)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process, let P be the available pool of resources of G, and let $\mathcal{H}_C$ be a complete history of G obtained by executing it. The* average resources usage *of G with regards to $\mathcal{H}_C$ is a n-tuple $((r_1, u_1), ..., (r_n, u_n))$ where $\forall i \in [1...n]$:*

- *$r_i \in P$ is a* resource*;*
- *$u_i = \frac{1}{|\mathcal{H}_C|} \sum_{t=0}^{|\mathcal{H}_C|-1} \frac{P(r_i) - \mathcal{H}_C[t][1](r_i)}{P(r_i)}$ is the* percentage of use *of resource r_i.*

Similarly to the resource usage of a task, the $\rho_{\mathcal{H}_C}(G)(r)$ operator can be used to obtain the average usage of a given resource r during the execution of G represented by $\mathcal{H}_C$.

Derived from the notion of resources usage, one can define the cost of executing a given process. This cost corresponds to the sum of the costs of the resources used by the process during its execution.

Definition 2.44 (Total Cost)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process, let P be the available pool of resources of G, and let $\mathcal{H}_C$ be a complete history of G obtained by executing it. The* total cost *of G with regards to $\mathcal{H}_C$ is a value defined as*

$$C_{\mathcal{H}_C} = |\mathcal{H}_C| \cdot \sum_{(r,n)\in P} P(r) \times \texttt{cost}(r)$$

Remark 2.9. *The shape of the total cost function comes from the fact that the cost per resources is considered the same whether the resource is used or not. Thus, the usage of each resource at each clock tick is not required to obtain the result.*

Another interesting metric, crucial in the work presented in this thesis, consists in measuring the waiting/synchronisation time of parallel merge gateways. Indeed, by definition, a parallel merge receives (and consumes) a token only when all its parent flows possess at least one token. However, there might be a non-negligible time gap between the time at which the first parent flow of the gateway receives a token, and the time at which the last one does. This duration, inducing delays in the execution of the process, is called the *synchronisation time* of parallel gateways.

Definition 2.45 (Synchronisation Time)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process and let $\mathcal{H}_C$ be a complete history of G obtained by executing it. For all $v \in V$ such that $\theta(v) = \diamondsuit_M$, the* synchronisation time *of v with regards to $\mathcal{H}_C$ is defined as*

$$ST_{\mathcal{H}_C}(v) \stackrel{\text{def}}{=} t_f - t_i$$

where

- $t_i = t \in [1...|\mathcal{H}_C|]$ *such that:*
  - $\exists v_p \in \texttt{parents}(v) \mid \mathcal{H}_C[t][0](v_p) \neq 0$
  - $\forall v_p \in \texttt{parents}(v) \mid \mathcal{H}_C[t-1][0](v_p) = 0$
- $t_f = t \in [1...|\mathcal{H}_C|]$ *such that:*
  - $\forall v_p \in \texttt{parents}(v) \mid \mathcal{H}_C[t][0](v_p) \neq 0$
  - $\exists v_p \in \texttt{parents}(v) \mid \mathcal{H}_C[t-1][0](v_p) = 0$

In the context of this work, BPMN processes may not be executed only once, but multiple times, each execution being called an *instance* of the process. The time separating the start of each instance of the process is called *rate*. The 2-tuple composed of the number of instances of the process being run, and the rate at which each of these instances will start is called a *workload*.

Definition 2.46 (Workload)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process. The* workload *of G is defined as a 2-tuple $W \stackrel{\text{def}}{=} (N, R)$ where:*

- $N \in \mathbb{N} \cup \{\infty\}$ *represents the* number of instances *of G being run;*

— $R \sim \mathcal{P}$ *represents the* rate *at which each new instance of* G *will start, following any probability distribution* $\mathcal{P}$*.*

Remark 2.10. *A workload* $W = (N, R)$ *is said to be a* closed workload *if* $N \neq \infty$*.*

In the rest of this thesis, only closed workloads will be considered, and the rate of a given workload will be called *inter-arrival time* (IAT). In this context, the execution time becomes the average of the execution times of the running instances, called *average execution time* (AET), the resources usage simply becomes the average usage of the resources over all the running instances, and the total cost of the process becomes the sum of the total costs of the running instances. Also, all the instances share a unique pool of resources, whose composition may drastically affect the aforementioned metrics.

All these indicators can be computed using simulation techniques [DRS19] based on the transition rules of processes executions defined in the previous section. In a few words, the simulation starts by performing the first task of the first instance of the process. If there are several parallel tasks to execute, it tries to execute all of them, based on the number of replicas of each of the resources required by these tasks. If there are not enough replicas of a resource, as many tasks as possible requiring that resource are non-deterministically chosen and executed. The ones that could not execute remain waiting. After a time R, another instance starts, complexifying the access to the shared resources, and the selection of the tasks that will execute. The simulator then considers those two instances running in parallel. The simulation ends when all the tokens send through the system have reached an end event.

2.5 Model Checking

Model checking [BK08] is a well-known technique consisting in verifying that a model represented as a transition system satisfies a given temporal logic property, which specifies some expected requirements of the system. One of the most common types of transition system used by model checkers is the *labelled transition system* [Mil89, Hoa78].

Definition 2.47 (Labelled Transition System)**.** *A* labelled transition system*, abbreviated* LTS*, is a 4-tuple* $M = (S, s^0, \Sigma, T)$ *such that:*

— S *is a finite set of* states*;*
— $s^0 \in S$ *is the* initial state*;*
— Σ *is a finite* set of actions*, or* labels*;*
— $T \subseteq S \times \Sigma \times S$ *is a finite* set of (labelled) transitions*.*

A transition is thus a 3-tuple (s, l, s')*, where* $l \in \Sigma$ *and* $s, s' \in S$*. For clarity, it is usually represented as* $s \xrightarrow{l} s'$*.*

An LTS can be viewed as all possible executions of a system. One specific execution is called a *trace*.

Definition 2.48 (LTS Trace)**.** *Let $M = (S, s^0, \Sigma, T)$ be an LTS. A* trace *of M is a sequence of labels $\lambda = (l_1, l_2, \ldots, l_n) \in \Sigma^n$ such that $\forall i \in [1...n],\ s^{i-1} \xrightarrow{l_i} s^i \in T$. The set of all traces of M is written Λ, and can be retrieved using the operator $\Lambda(M)$.*

The verification of such a system is usually done by formalising the expected behaviour of that system with a set of temporal logic properties. Several temporal logics exist in the literature, and are usually divided into two families: linear time logics, which assume that time is linear, i.e., there is only one possible future at each instant of time, and branching-time logics, which assume that at each moment, time may split into alternate courses representing different possible futures [EH83]. Among them, we can cite several famous ones such as the Linear Temporal Logic (LTL) [Pnu77], the Computation Tree Logic (CTL) [CE82], CTL* [EH83], or the Model Checking Language (MCL) [MT08]. Temporal logic properties are themselves usually divided into two distinct families: safety and liveness properties [BK08]. Safety properties state that something bad must never happen while liveness properties state that something good will eventually happen. In this thesis, we focus on the Linear Temporal Logic because it is rather simple to use and expressive enough to represent properties on execution paths.

The LTL syntax consists of propositions, logical operators ($\neg$, $\vee$, $\Rightarrow$, ...), and temporal operators (**X**, **U**, **G**, **F**, **W**, ...). For two LTL properties φ and ψ, and a state s of the model, the meaning of these operators is defined as:

— $\neg\varphi$ holds in s if φ does not;
— $\varphi \vee \psi$ holds in s if φ holds or if ψ holds;
— $\varphi \Rightarrow \psi$ holds in s if $\neg\varphi$ holds or if ψ holds;
— **X** φ requires φ to hold in the state following s;
— φ **U** ψ requires φ to hold in s and all its future states until reaching a state s' in which ψ holds;
— **G** φ requires φ to hold in s and all its future states;
— **F** φ requires φ to hold in s or (at least) one of its future states;
— φ **W** ψ requires φ to hold in s and all its future states or to hold until reaching a state s' in which ψ holds.

When a model checker is given as input a model M and a temporal logic property φ, it verifies whether this property holds on the model or not. If the property is satisfied, i.e., if it holds for every trace of the model, written $M \models \varphi$, it simply returns *True*. Otherwise, if the property is violated, written $M \not\models \varphi$, it returns *False*, and usually provides a *counterexample* of that property. In this thesis, a counterexample is a trace of the LTS that does not satisfy the given property.

Definition 2.49 (Counterexample)**.** *Let $M = (S, s^0, \Sigma, T)$ be an LTS and let φ be a temporal logic property such that $M \not\models \varphi$. A* counterexample *of M is a trace $\lambda \in \Lambda(M)$ such that $\lambda \not\models \varphi$.*

Chapter 3

Modelling BPMN Processes from Textual Requirements

"If you can't describe what you are doing as a process, you don't know what you're doing."

William Edwards Deming

Contents

Modelling business processes is a common task that companies perform in order to obtain a graphical representation of their business processes. However, this is a challenging task, as it can quickly become tedious and error-prone. On the one hand, it usually requires a deep understanding of the company's needs and of its intrinsic way of functioning. Consequently, this task often becomes time-consuming too. On the other hand, it necessitates strong competences in modelling, and in particular an advanced knowledge of the BPMN notation. Indeed, the BPMN modelling tools allow a lot of freedom in the design of the processes, and usually do not provide integrated solutions for asserting their correctness. Thus, a novice user can quite easily design a syntactically or semantically incorrect process. Moreover, complex constructs, such as unbalanced gateways or nested loops, may be challenging to handle for inexperienced users. For these multiple reasons, we propose an approach aiming at automatically generating a business process in the BPMN notation from a textual representation of it, while providing strong guarantees to its semantics.

The approach proposed in this chapter only requires a textual description of the process-to-be, written in natural language. To gain in accuracy, the description may (but is not required to) provide names to the tasks that should appear in the process. If the tasks that should appear in the process are already named, the description is sent to a fine-tuned version of GPT, which is in charge of extracting all the relationships, or *ordering constraints*, that may exist between them. These constraints, which describe the relationships of the tasks with regards to their siblings, are decomposed in four categories: (i) the *sequential constraints*, symbolising the fact that some tasks should be executed before some others, (ii) the *mutual exclusion constraints*, symbolising the fact that some tasks are mutually exclusive of some others, (iii) the *(explicit) loop constraints*, symbolising the fact that some tasks can be repeated, and (iv) the *parallel constraints*, symbolising the fact that some tasks can be executed in parallel. As a result, GPT returns a set of expressions compliant with a grammar that we defined, and which encompasses the behaviour of all the business processes that can be written considering the excerpt of the BPMN syntax presented in Figure 2.1. However, we said that the user was not required to name the tasks of the process, and could pursue with the original, "raw" description. In this case, the description is first sent to the classical version of GPT (currently, GPT-4.5-preview), which is asked to analyse the description, infer some tasks from it, name them, modify the description to replace the portions of text corresponding to these tasks by their name, and return the modified description. This new description is then given to the fine-tuned version of GPT.

Generating a BPMN process from a set of expressions representing the ordering constraints that must exist between its tasks mostly consists in gathering the information sparsed in these expressions in a structure that can eventually be converted into a BPMN process. The structure chosen in this approach is a graph, as it can easily be mapped to the corresponding BPMN process. The graph is built by analysing the sequential constraints belonging to the expressions returned by GPT. It is then converted to BPMN by introducing the miss-

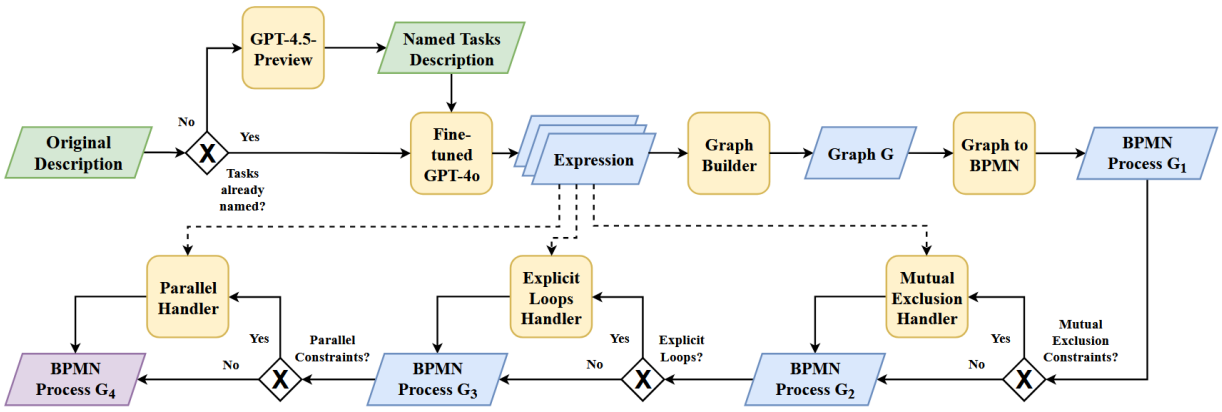


Figure 3.1: Overview of the Approach

ing control flow elements (events and gateways), while preserving its semantics. Finally, the three other possible types of constraints, namely, the mutual exclusion constraints, the explicit loops, and the parallel constraints, are considered, and their behaviour is incorporated into the BPMN process. The generated BPMN process is eventually returned to the user. These steps are recalled in Figure 3.1.

This approach was fully implemented and consists of approximately 12k lines of Java code. Details about its implementation and its evaluation will be given in Section 5.1 of this manuscript.

3.1 Textual Requirements

In this work, the user must provide as input a textual representation of a business process that is going to be generated. This description is informal, in the sense that no prerequisite is required to be able to write a valid one. In this description, the user expresses in natural language the tasks that the process should contain, along with their ordering constraints. For instance, if two tasks must be executed one after the other in the final process, this should be stated in the description. To improve the results, the user is advised to name the tasks that should appear in the process. However, this is not mandatory for the approach to work.

Running example. Let us propose as running example the textual description of a software feature management process: "First, the developer *StartFeatureManagementSoftware (StFMS)*. Then, he *DescribeNewFeatureRequirements (DNFR)*. After that, the staff *ValidateInternally (VI)*, and the client *ValidateExternally (VE)*. Once the feature has been validated internally, the developer can *CreateNewFeatureBranch (CNFB)*. Once the feature is completely validated (internally and externally), the staff can *StartTechnicalDesign (STD)*. Instead of describing a new feature, validate it, create a new branch and start technical design, the developer can also *LoadCurrentlyDevelopedFeature (LCDF)*. The *Fea-*

tureDevelopment (FD) then eventually starts, followed by a *DebuggingPhase (DP)* useful to chase possible bugs before releasing the feature. This phase leads either to a *BugCaseOpening (BCO)*, or to *ReleaseFeature (RF)* if no bug was found. If a bug case is opened, three different operations may start: either the first support level initiates a *FirstStageDebugPhase (FSDP)*, which eventually leads to *ClosingFirstLevelRequest (CFLR)*, or the second support level initiates a *SecondStageDebugPhase (SSDP)*, which eventually leads to *ClosingSecondLevelRequest (CSLR)*, or the third support level initiates a *ThirdStageDebugPhase (TSDP)*, which eventually leads to *ClosingThirdLevelRequest (CTLR)*. Once these phases are closed, either there is no bug anymore to correct, and the *ReleaseFeature* task *(RF)* occurs, or a new bug is found, leading to *DebuggingPhase (DP)* again. Also, the *FirstStageDebugPhase (FSDP)*, *SecondStageDebugPhase (SSDP)* and *ThirdStageDebugPhase (TSDP)* and their closing can be repeated until a bug is properly corrected. Once *ReleaseFeature (RF)* occurred, the developer can either *ShutdownFeatureManagementSoftware (ShFMS)*, or start again with the *DescribeNewFeatureRequirements* task *(DNFR)*."

As the reader can see, the specification is rather informal, except that names are given to the tasks that should appear in the process. The specification can be written in various styles, with or without context surrounding the important information. The acronyms written between parenthesis are not mandatory, and are presented here to shorten the size of the future examples.

3.2 Task Ordering Constraints

To generate a BPMN process from a textual description, one of the intermediate steps consists in extracting some *task ordering constraints* from the text. These constraints represent the way in which the tasks that should appear in the process are related to each others. Based on the supported fragment of the BPMN syntax presented in Figure 2.1, we defined 5 different types of constraints between tasks: (i) the sequential constraint, expressing the fact that two tasks must be executed in a certain order, one after the other, (ii) the mutual exclusion constraint, expressing the fact that, if one of the two tasks is executed, then the other one should not be, (iii) the parallel constraint, expressing the fact that two tasks can be executed at the same time, (iv) the looping constraint, expressing the fact that two tasks can be repeated, and (v) the absence of constraints, expressing the fact that two tasks are not constrained to each other.

To analyse such constraints, one needs to express them in a systematic way removing any ambiguity that may appear in their natural language description. In this work, we rely on a simple language defining *operators* for the five types of constraints presented above. These operators are:

— the '$<$' operator, handling the sequential constraint (associative, not commutative, not idempotent)
— the '|' operator, handling the mutual exclusion constraint (associative, commutative,

idempotent)
— the '&' operator, handling the parallel constraint (associative, commutative, idempotent)
— the '$*$' operator, handling the looping constraint (associative, commutative, idempotent)[1]
— the ',' operator, handling the absence of constraint (associative, commutative, idempotent)

To be able to specify constraints in this language, one must obey the rules of the following Backus-Naur Form (BNF) grammar:[2]

$$\begin{array}{lllllllllll} \langle \mathrm{E} \rangle & ::= & \texttt{t} & | & (\langle \mathrm{E} \rangle) & | & \langle \mathrm{E}_1 \rangle\ \langle \mathrm{op} \rangle\ \langle \mathrm{E}_2 \rangle & | & (\langle \mathrm{E}_1 \rangle)* \\ \langle \mathrm{op} \rangle & ::= & \text{`}|\text{'} & | & \text{`\&'} & | & \text{`}<\text{'} & | & \text{`,'} \end{array}$$

where `t` is a terminal symbol representing (the name of) a task of the process.

To remove any possible ambiguity from this language, we must also provide *priority* between its operators. Indeed, without such a notion, one could not state whether the expression $\langle \mathrm{E}_1 \rangle < \langle \mathrm{E}_2 \rangle \mid \langle \mathrm{E}_3 \rangle$ is equivalent to the expression $(\langle \mathrm{E}_1 \rangle < \langle \mathrm{E}_2 \rangle) \mid \langle \mathrm{E}_3 \rangle$, or to the expression $\langle \mathrm{E}_1 \rangle < (\langle \mathrm{E}_2 \rangle \mid \langle \mathrm{E}_3 \rangle)$. By abusively borrowing the '$\prec$' symbol from the partial order notation, the priorities between the operators of this language can be written this way: '$*$' $\prec$ ',' $\prec$ '$<$' $\prec$ '&' $\prec$ '$|$' . Consequently, one can now state that expression $\langle \mathrm{E}_1 \rangle < \langle \mathrm{E}_2 \rangle \mid \langle \mathrm{E}_3 \rangle$ is equivalent to expression $(\langle \mathrm{E}_1 \rangle < \langle \mathrm{E}_2 \rangle) \mid \langle \mathrm{E}_3 \rangle$ (the '$|$' operator having a higher priority than the '$<$' operator). In the rest of this work, the set of expressions returned by GPT will be called `Expr`. Similarly to graphs paths, the operator `tasks`(e) will be used to retrieve all the tasks belonging to an expression $e \in$ `Expr`.

Example. Let us consider the textual description of the running example. By analysing it, one can generate the following set of expressions `Expr`, containing 10 expressions capturing all the tasks ordering constraints belonging to the description:

(1) $StFMS < DNFR < (VI,\ VE)$
(2) $VI < CNFB$
(2) $(VI,\ VE) < STD$
(4) $(DNFR,\ VI,\ VE,\ CNFB,\ STD) \mid LCDF$
(5) $(STD,\ CNFB) < (FD < DP)$
(6) $DP < (BCO \mid RF)$
(7) $BCO < ((FSDP < CFLR) \mid (SSDP < CSLR) \mid (TSDP < CTLR))$
(8) $(CFLR,\ CSLR,\ CTLR) < (RF \mid DP)$

[1] We will see that, due to the definition of the grammar, these notions can inherently not be applied to this operator.
[2] This is why the '$*$' operator's associativity/commutativity/idempotence is not useful.

(9) (*FSDP*, *SSDP*, *TSDP*, *CFLR*, *CSLR*, *CTLR*)∗
(10) *RF* < (*ShFMS* | *DNFR*)

As an illustration, expression (7) means that task *BCO* must be executed before tasks *FSDP*, *CFLR*, *SSDP*, *CSLR*, *TSDP*, and *CTLR*. Moreover, it also means that task *FSDP* must be executed before task *CFLR*, task *SSDP* must be executed before task *CSLR*, and task *TSDP* must be executed before task *CTLR*. Finally, it also means that tasks *FSDP* and *CFLR* are mutually exclusive of tasks *SSDP* and *CSLR*, which are themselves mutually exclusive of tasks *TSDP* and *CTLR*.

3.3 Fine-tuning GPT-4o

Inferring ordering constraints from a textual description is not a trivial task, as it requires complex mechanisms to understand the structure of the text, extract its components (i.e., the tasks), and discover the possible relationships connecting them. In this work, Large Language Models (LLMs)—and more precisely, the GPT model [ea24b]—are used to perform this analysis. GPT, which stands for Generative Pre-trained Transformer, is an open-access generative model developed by OpenAI, and freely accessible through the well-known website ChatGPT[3]. GPT, and more precisely its "4o" version, is used in this work for its natural language processing capabilities, that are helpful to extract expressions corresponding to the language presented in Section 3.2 from a textual description.

The standard version of the GPT-4o model (GPT, as a shorthand, in the rest of this manuscript) has no knowledge about the expected format that its output should take, that is, the language defined in Section 3.2. Thus, it is not straightforward for it to generate expressions compliant with this language. To mitigate such issues, GPT can be fine-tuned [WCF+25] in order to increase its capabilities in precise fields. Roughly speaking, fine-tuning is an approach consisting in improving the capacities of a model on a specific field by training it on a well-defined set of new data. Often, fine-tuning becomes a tedious task as adjusting hyper-parameters and providing sufficient data can be rather complex and time-consuming. Hopefully, GPT proposes some intuitive and easy-to-follow fine-tuning options, which do not require large amounts of data to get started. This phase, which can be repeated at any time to improve the quality of the results, has been performed on approximately four hundred examples in the context of this thesis. It is worth noting that the fine-tuning phase was performed on the "4o" version of GPT, which was the latest version of GPT available for fine-tuning at the moment this work has been done.

The training examples provided to GPT consist of three elements. The first one is a system prompt, which describes the expected behaviour of GPT (i.e., the fact that GPT should extract task ordering constraints from the textual requirements given as input) and the shape that its output should take. The second one is a user prompt, usually corresponding to the question asked by the user (i.e., the textual requirements here). The

[3]https://chatgpt.com/

last one is an assistant prompt, corresponding to the answer that GPT should provide. For the system prompt and the assistant prompt, the expected output is a set of expressions corresponding to the language defined in Section 3.2. For all of them, an example is provided in Appendix A.

Once training and validation data are given to GPT, the fine-tuning process starts automatically. All over its fine-tuning, GPT provides metrics indicating how good the training is going. The reader interested in them can take a look at Appendix B. When the fine-tuning finishes, a new version of the base model is generated and made available to the user. After this fine-tuning phase, the altered model was able to transform textual requirements into expressions. It is worth noting that GPT may generate several expressions to represent all the task ordering constraints that it finds in a given description. It is for instance the case in the running example, where the description states that: "After that, the staff *ValidateInternally (VI)*, and the client *ValidateExternally (VE)*. Once the feature has been validated internally, the developer can *CreateNewFeatureBranch (CNFB)*. Once the feature is completely validated (internally and externally), the staff can *StartTechnicalDesign (STD)*". These information have to be split into two expressions: $VI < CNFB$ and $(VI,\ VE) < STD$, as shown in the example of Section 3.2. Indeed, a single expression, such as $(VI,\ VE) < (CNFB,\ STD)$ would introduce the unspecified constraint $VE < CNFB$.

3.4 Parsing & Refinement of Expressions

3.4.1 Expressions as Arborescences

The first step towards the construction of the BPMN process from the given set of expressions consists in parsing these expressions, and transforming them into a computer-friendly structure. In this work, this structure is an *arborescence*, which is a tree structure enhanced with a *root*, and *directions* for its edges. Parsing an expression and producing the corresponding arborescence can easily be achieved with classical parsing techniques [GJ07]. As a result, one obtains a binary ordered arborescence $T_{\preceq} = (V, E, R)$ hierarchically representing the semantics of the expression, whose vertices are operators of the language defined in Section 3.2, and whose root is the operator of highest priority appearing in the expression. Let us now define a *convention for the arborescences* used in this thesis, along with several useful *operators* that will be used throughout this section.

Definition 3.1 (Arborescences Convention)**.** *Let $T = (V, E, R)$ be an arborescence. We define:*

— (t) *as a vertex $v \in V$ representing a* task*;*
— (,) *as a vertex $v \in V$ representing the* ',' *operator;*
— (&) *as a vertex $v \in V$ representing the* '&' *operator;*

— $(<)$ *as a vertex* $v \in V$ *representing the* '$<$' *operator;*
— $(|)$ *as a vertex* $v \in V$ *representing the* '$|$' *operator;*
— $(*)$ *as a vertex* $v \in V$ *representing the* '$*$' *operator;*
— (T_i) *as a vertex* $v \in V$ *representing a* (sub-)arborescence T_i *compliant with the above convention.*

Definition 3.2 (Arborescences Operators)**.** *Let* $T = (V, E, R)$ *be an arborescence. We define:*

— *For all* $v \in V$, $\theta(v) \in \{\texttt{t}, \text{','}, \text{'\&'}, \text{'}<\text{'}, \text{'}|\text{'}, \text{'}*\text{'}\}$, *which returns the* type of the vertex v*;*
— $\texttt{tasks}(T) \stackrel{\text{def}}{=} \{v \in V \mid \theta(v) = \texttt{t}\}$ *which returns the* set of tasks *belonging to* T*.*

Remark 3.1. *As the defined language is associative, it is consequently right-associative and left-associative. Thus, one can choose one or the other type of associativity without any loss of generality. This choice will only impact the form of the generated arborescence (right or left unbalancing, if any) and its structure (the arborescence is a binary arborescence).*

So as to be handled properly, the relationship between arborescences and expressions must be a *1-to-1 mapping*, meaning that each expression must have its corresponding arborescence, and that each generated arborescence must correspond to an expression.

Definition 3.3 (Expressions and Arborescences Mapping)**.** *Let* `Expr` *be a set of expressions, and* $S_{T_\preceq}$ *be a set of binary ordered arborescences generated from these expressions. By abusively borrowing the* '$\models$'*symbol from mathematical logic, we can write that:*

— *For all* $e \in \texttt{Expr}$*, there exists* $T_\preceq \in S_{T_\preceq}$ *such that* $T_\preceq \models e$*;*
— *For all* $T_\preceq \in S_{T_\preceq}$*, there exists* $e \in \texttt{Expr}$ *such that* $e \models T_\preceq$*.*

Example. Considering the 10 expressions of the running example, one can build the 10 binary ordered arborescences shown in Figure 3.2. Arborescence (1) illustrates the fact that we chose right-associativity, as its root node corresponds to the first '$<$' operator of expression (1). In case of left-associativity, the root node would have corresponded to the second '$<$' operator of the expression.

3.4.2 Reduction of Arborescences

To simplify the traversal of these arborescences, one can apply an operation called *reduction* on them, which minimises their size, while preserving their semantics.

Definition 3.4 (Ordered Arborescence Reduction)**.** *Let* $T_\preceq = (V, E, R)$ *be an ordered arborescence. The* reduction operation *can be applied to any vertex* $v \in V$, $\theta(v) \neq \texttt{t}$*, such that there exists* $v_i \in \texttt{childs}(v)$ *for which* $\theta(v) = \theta(v_i)$*. In such a case, the reduction operation performs the following changes:*

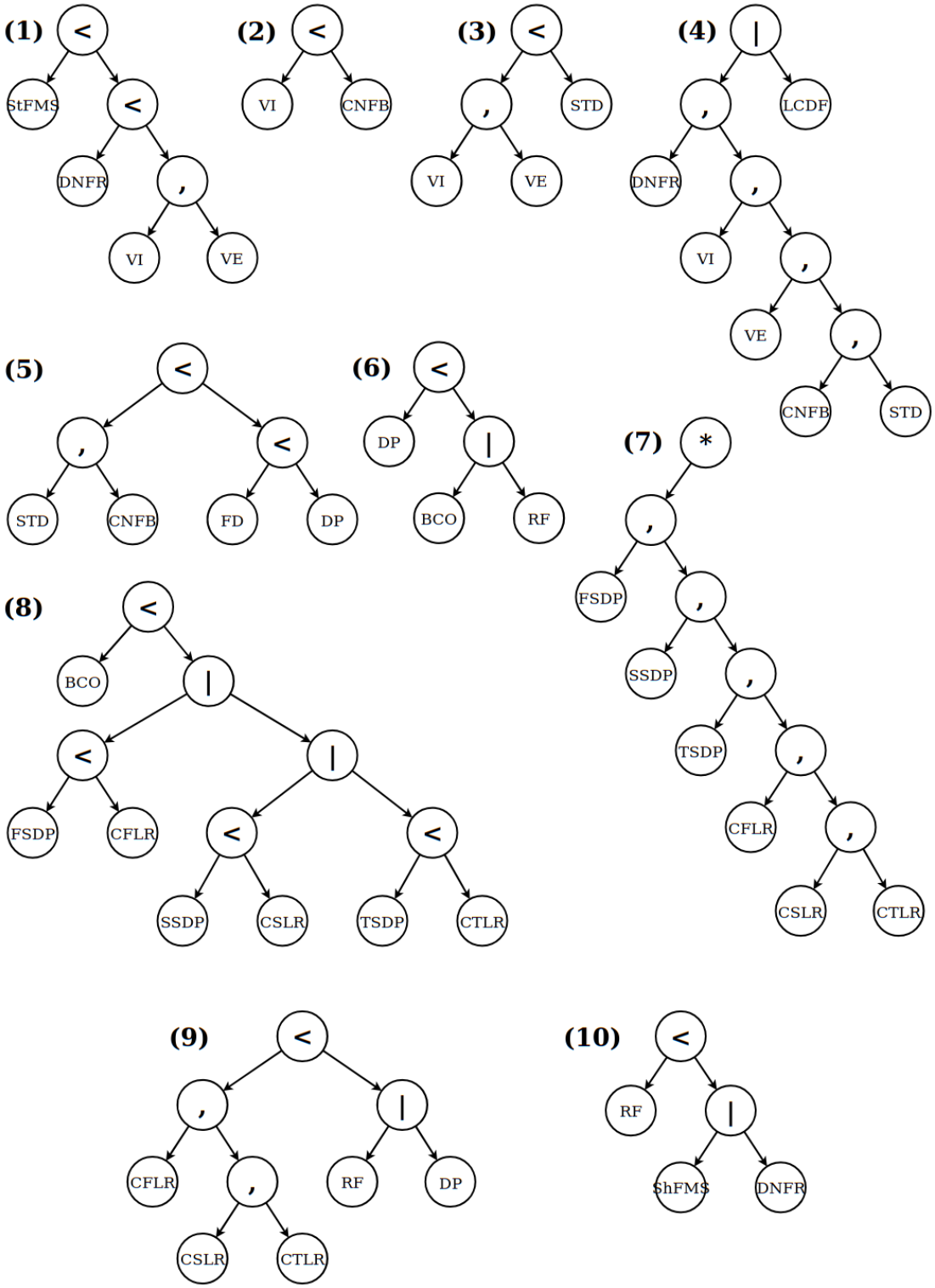


Figure 3.2: Original Arborescences Generated from the Constraints of the Running Example

— *For all* $v_j \in \texttt{childs}(v)$ *such that* $j > i$, $j \leftarrow j + |\texttt{childs}(v_i)| - 1$ *(right shifting of the indices to make room for the new nodes);*
— $V = V \setminus \{v_i\}$ *(removal of the no longer useful node);*
— *For all* $v_k \in \texttt{childs}(v_i)$, $k \leftarrow k + i$ *(update of the old children of* v_i*'s indices);*
— $E = E \setminus \{v \rightarrow v_i\} \setminus \bigcup_{v_k \in \texttt{childs}(v_i)} \{v_i \rightarrow v_k\} \cup \bigcup_{v_k \in \texttt{childs}(v_i)} \{v \rightarrow v_k\}$ *(switching of the edges).*

An ordered arborescence $T'_{\preceq} = (V', E', R)$ *obtained by applying the aforementioned reduction is said to be* reduced *if and only if for all* $v \in V'$, *for all* $v' \in \texttt{childs}(v)$, $\theta(v) \neq \theta(v')$, *and* partially reduced *if* $T'_{\preceq}$ *is not reduced and is not a binary directed arborescence. The degree of reduction of* $T'_{\preceq}$ *is defined as* $\texttt{deg}(T'_{\preceq}) \stackrel{\text{def}}{=} |V| - |V'|$.

Proposition 3.1 (Semantics Preservation)**.** *Let* $T_{\preceq} = (V, E, R)$ *be an ordered arborescence, and let* $T'_{\preceq} = (V', E', R)$ *be the reduced version of* $T_{\preceq}$. *We state that* $T_{\preceq}$ *and* $T'_{\preceq}$ *are semantically equivalent, in the sense that the expressions corresponding to* $T_{\preceq}$ *and* $T'_{\preceq}$ *are strictly identical.*

Proof. Let us consider a binary directed arborescence $T_{\preceq}$, and let $T^n_{\preceq}$ be a partially reduced version of $T_{\preceq}$ of reduction degree n. Let us reason by structural induction on the degree of reduction of the arborescence, and with the hypothesis that $T_{\preceq}$ and $T^n_{\preceq}$ are semantically equivalent. We will show that the partially reduced version of $T_{\preceq}$ of reduction degree $n+1$, namely $T^{n+1}_{\preceq}$, is semantically equivalent to $T^n_{\preceq}$, and thus to $T_{\preceq}$.

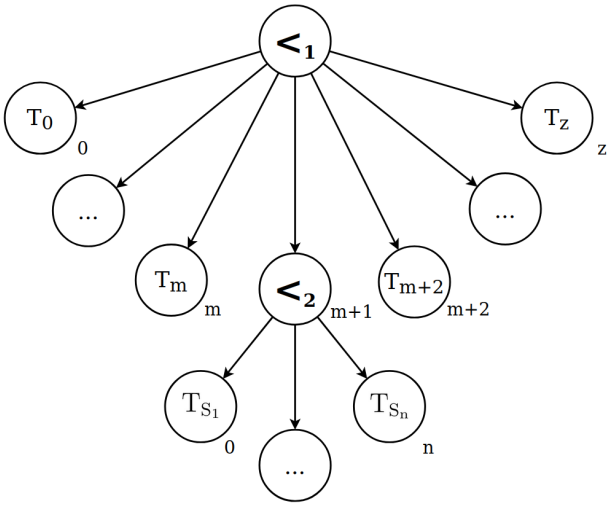
— Base case: The empty ordered arborescence $T^0_{\preceq}$ and its reduced version $T'^0_{\preceq}$ are trivially semantically equivalent.

— Induction: Let us consider without loss of generality that $T^n_{\preceq}$ corresponds to the arborescence shown in Figure 3.3(a), and $T^{n+1}_{\preceq}$ to the arborescence shown in Figure 3.3(b). $T^n_{\preceq}$ has the following properties:

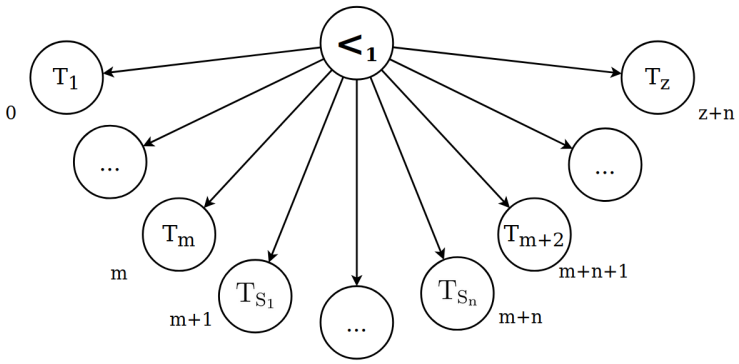
(1) $\forall i \in [1...n],\ T_m \preceq T_{S_i} \ \wedge \ T_{S_i} \preceq T_{m+2}$ (by transitivity of the partial order relationship);
(2) $\forall i \in [1...n-1],\ T_{S_i} \preceq T_{S_{i+1}}$;
(3) $\forall i \in [0...m-1],\ T_i \preceq T_{i+1}$;
(4) $\forall i \in [m+2...z-1],\ T_i \preceq T_{i+1}$;
(5) $\forall i \in [1...z],\ \theta(\texttt{parent}(T_i)) = \text{`}<\text{'}$;
(6) $\forall i \in [1...n],\ \theta(\texttt{parent}(T_{S_i})) = \text{`}<\text{'}$.

We show easily that the same properties still hold in $T^{n+1}_{\preceq}$:

(1) remained true thanks to the indices shift;
(2) remained true thanks to the indices shift;
(3) remained true trivially;
(4) remained true thanks to the indices shift;



(a) Ordered Arborescence of Reduction Degree n



(b) Ordered Arborescence of Reduction Degree $n+1$

Figure 3.3: Ordered Arborescences with Different Reduction Degree

(5) remained true trivially;
(6) remained true trivially.

Thus, $T_{\preceq}^{n+1}$ is semantically equivalent to $T_{\preceq}^{n}$, and, by induction hypothesis, to $T_{\preceq}$.

□

Example. Figure 3.4 shows the result of applying the reduction operation on the 10 original arborescences. As the reader can see, the root node of arborescence 1 now has 3 successors: *StFMS*, *DNFR*, and a ',' node.

3.4.3 Useful Operators and Sets

Finally, we define the set of (explicit) loops of the process, along with two operators returning respectively the mutually exclusive tasks of a given task and the parallel tasks

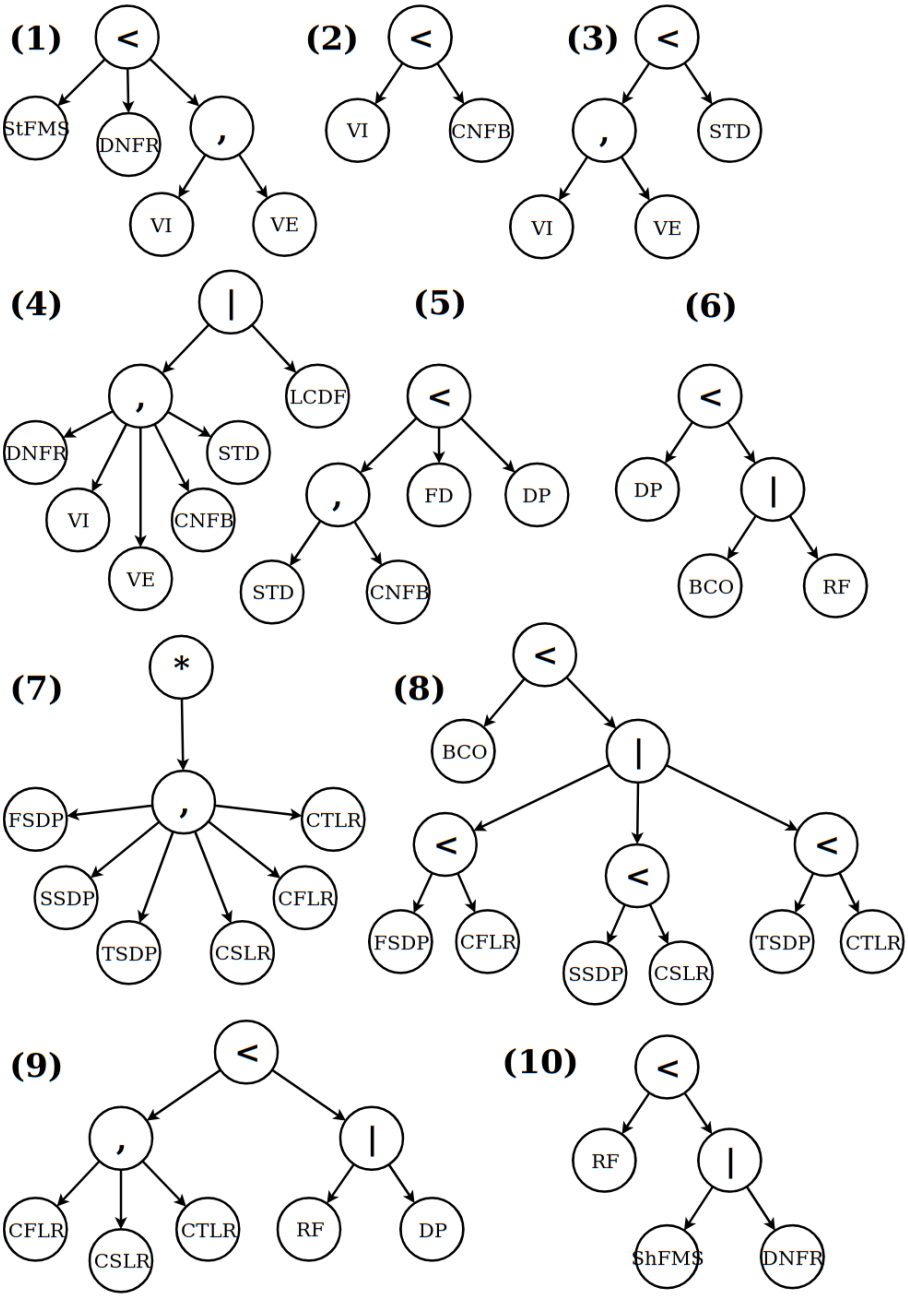


Figure 3.4: Reduced Versions of the Arborescences Shown in Figure 3.2

of a given task.

Definition 3.5 ((Explicit) Loops Set)**.** *Let* `Expr` *be a set of expressions. The* set of explicit loops *of* `Expr` *is defined as*

$$\texttt{Loops} \stackrel{\text{def}}{=} \bigcup_{e \in \texttt{Expr}} \{(v_1, ..., v_n) \in \texttt{tasks}(e) \mid e = (e')*\}.$$

Definition 3.6 (Mutual Exclusions Operator)**.** *Let* $S_{T_\preceq}$ *be a set of ordered arborescences representing a set of expressions* `Expr`*. For all* $T_\preceq = (V, E, R) \in S_{T_\preceq}$*, for all* $v \in V$ *such that* $\theta(v) = \textcircled{t}$*, we define the* mutual exclusions operator *as*

$$\texttt{mutex}(v) \stackrel{\text{def}}{=} \bigcup_{T'_\preceq = (V', E', R') \in S_{T_\preceq}} \{v' \in V' \mid \theta(v') = \textcircled{t} \ \wedge \ \texttt{m}(v, v')\}$$

where

$$\texttt{m}(v, v') \Leftrightarrow \exists v'' \in V', \theta(v'') = \textcircled{|}, \ \exists v_1, v_2 \in \texttt{childs}(v''), \ v_1 \neq v_2, \mid v \in T_\preceq(v_1) \wedge v' \in T_\preceq(v_2).$$

Definition 3.7 (Parallel Operator)**.** *Let* $S_{T_\preceq}$ *be a set of ordered arborescences representing a set of expressions* `Expr`*. For all* $T_\preceq = (V, E, R) \in S_{T_\preceq}$*, for all* $v \in V$ *such that* $\theta(v) = \textcircled{t}$*, we define the* parallel operator *as*

$$\texttt{par}(v) \stackrel{\text{def}}{=} \bigcup_{T'_\preceq = (V', E', R') \in S_{T_\preceq}} \{v' \in V' \mid \theta(v') = \textcircled{t} \ \wedge \ \texttt{p}(v, v')\}$$

where

$$\texttt{p}(v, v') \Leftrightarrow \exists v'' \in V', \theta(v'') = \textcircled{\&}, \ \exists v_1, v_2 \in \texttt{childs}(v''), \ v_1 \neq v_2, \mid v \in T_\preceq(v_1) \wedge v' \in T_\preceq(v_2).$$

3.5 Construction of a Graph handling Sequential Constraints

The first computational step of this approach consists in representing the sequential constraints appearing in the expressions of `Expr` on a graph. By definition, given a graph $G = (V, E, \Sigma)$, a node $v_1 \in V$ that is connected to another node $v_2 \in V$ such that $v_1 \rightarrow v_2 \in E$ models a sequential constraint between these two nodes, as v_1 has to be executed before v_2. This sequential constraint matches exactly the behaviour of the '<' operator. Thus, by considering all the expressions containing this '<' operator, one can build a graph $G = (V, E, \Sigma)$ handling all the sequential constraints, named $Cons_1$. To do so, the construction is based on an analysis of the arborescences corresponding to the

expressions. However, this analysis must be done carefully in order to avoid inserting too much information (i.e., too many edges), or too few, to the graph.

3.5.1 Simple Construction of the Graph

Considering the set of arborescences S_T corresponding to the expressions `Expr`, a (simple) way to generate a graph that represents its sequential constraints is to traverse each arborescence in a depth-first way, and create an edge between each task belonging to the subtree of a sequential vertex, and each task belonging to the immediately next subtree of this sequential vertex. More formally, we can create a graph $G = (V, E, \Sigma)$ where:

— $V = \bigcup_{T=(V_T,E_T,R_T)\in S_T} \bigcup_{\substack{v\in V_T \\ \theta(v)=\text{`<'}}} \bigcup_{v_c\in \texttt{childs}(v)} \texttt{tasks}(T_{\preceq}(v_c))$;

— $E = \bigcup_{T=(V_T,E_T,R_T)\in S_T} \bigcup_{\substack{v\in V_T \\ \theta(v)=\text{`<'}}} \bigcup_{i\in[0...|\texttt{childs}(v)|-1]} \bigcup_{v_1\in T_i} \bigcup_{v_2\in T_{i+1}} \{v_1 \to v_2\}$
where $T_i = \texttt{tasks}(T_{\preceq}(\texttt{childs}(v)[i]))$;

— $\Sigma = \bigcup_{v\in V} \{\sigma(v)\}$.

However, this simple construction may generate unnecessary, or, in the worst case, incorrect edges.

Example. Considering the expressions given in Section 3.2, this simple method would generate the graph shown in Figure 3.5. This method generates several unnecessary and/or incorrect edges, corresponding to the red dashed edges of the graph. For instance, due to the structure of the tree representing expression (7), task *BCO* will be connected to both tasks *FSDP* and *CFLR*. However, the connection to task *CFLR* is unnecessary, because task *FSDP* already precedes task *CFLR*, and even wrong, as the description states that "*This phase leads either to a BugCaseOpening (BCO), or to ReleaseFeature (RF) if no bug was found. If a bug case is opened, three different operations may start: either the first support level initiates a FirstStageDebugPhase (FSDP), which eventually leads to ClosingFirstLevelRequest (CFLR)* [...]". Thus, task *BCO* should not be able to reach task *CFLR* before performing task *FSDP*.

3.5.2 Transitive Reduction of the Graph

To avoid such incorrect connections, a solution could be to reduce the graph G by using classical transitive reduction algorithms [AGU72].

Definition 3.8 (Transitive Reduction of a Graph)**.** *Let $G = (V, E, \Sigma)$ be a graph.* A transitive reduction of G *is a graph $G' = (V, E', \Sigma)$ such that*

$$E' = \{e = v \to v' \in E \mid \nexists (e_0 = v \to v_1, e_1 = v_1 \to v_2, ..., e_n = v_n \to v') \in E\}$$

Said differently, G' only contains the longest paths connecting any two nodes of G.

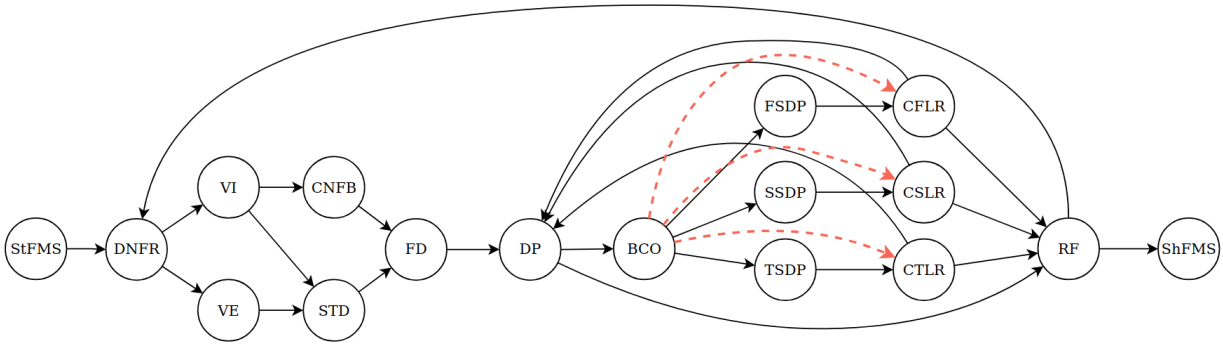


Figure 3.5: Graph Generated by the Simple Construction Algorithm

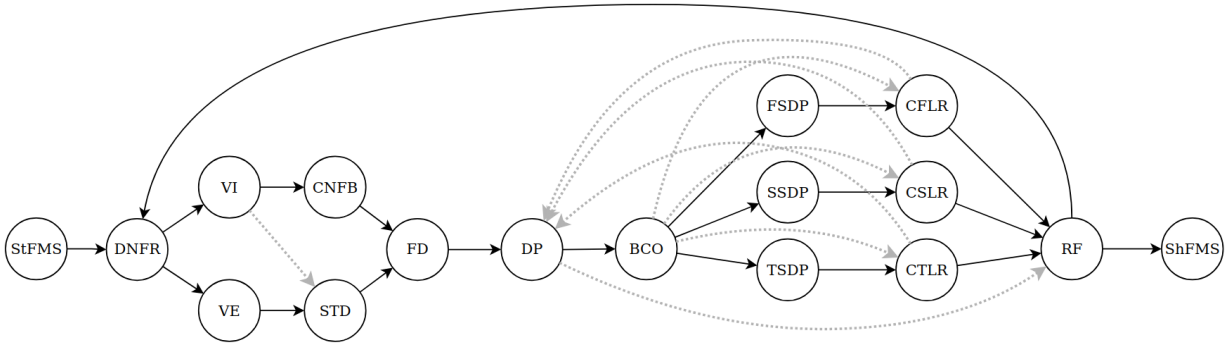


Figure 3.6: Transitively Reduced Version of the Graph Shown in Figure 3.5

The transitive reduction of G, performed in a breadth-first way, generates a graph G' containing less edges than G. Moreover, it is unlikely to contain the aforementioned unnecessary and/or incorrect edges. However, the transitive reduction may remove transitions considered as unnecessary in the sense of Definition 3.8, although being meaningful and mandatory in our case. Such bad removals are even more frequent in cyclic graphs, where several paths belonging to the cycle will be considered as useless and thus removed by the transitive reduction. Thus, transitively reducing G is not a good option in the context of our process generation.

Example. For instance, let us consider Figure 3.6, which corresponds to the transitive reduction of the graph given in Figure 3.5. As one can see, several flows have disappeared in this new version (they are materialised by the gray dotted lines in the figure). Notably, there is no longer a transition connecting nodes DP and RF, due to the fact that the path $p = DP \rightarrow BCO \rightarrow TSDP \rightarrow CTLR \rightarrow RF \in \mathcal{P}_G$, and that $|p| > |DP \rightarrow RF|$. However, this transition was meaningful in our context, as expression (6) depicts the fact that, after performing task DP, one can perform either task BCO or task RF. Thus, the transitive reduction of the graph is not a valid representation of the sequential constraints given in the expressions.

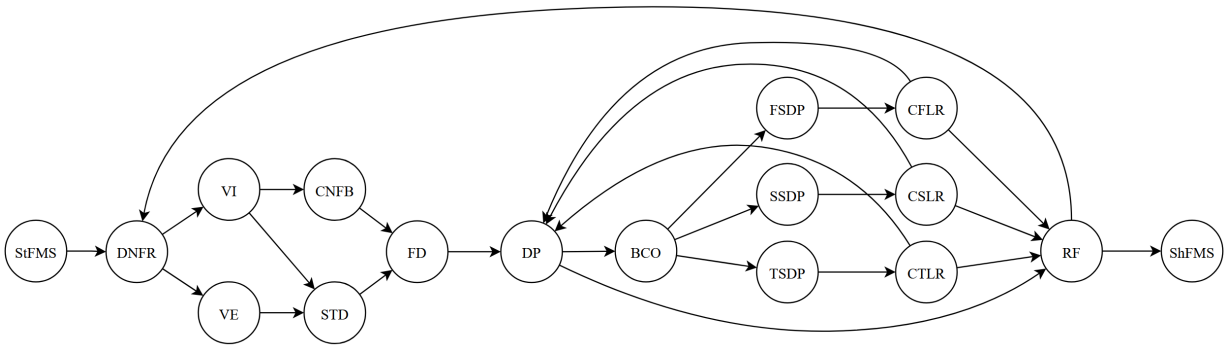


Figure 3.7: Graph Representing the Strong Sequential Constraints of `Expr`

3.5.3 Smart Construction of the Graph

To generate a graph compliant with the expressions returned by GPT, our proposal consists in keeping only the *strong sequential constraints* given in the expressions. These constraints depict mandatory edges of the graph that order vertices which would not be in any sequential order without them. The extraction of such constraints is done by Algorithm 1, which traverses the arborescences corresponding to the expressions in a depth-first way, and, each time it finds a sequential node, creates new strong sequential constraints consisting of pairs containing the rightmost tasks of the i^{th} subtree of the sequential node, and the leftmost tasks of the $i + 1^{th}$ subtree of the sequential node. These strong sequential constraints are then used to build the graph G, in a similar manner to that presented in Section 3.5.1.

Example. The graph generated from the (strong) sequential constraints extracted by Algorithm 1 from the arborescences representing the expressions `Expr` is shown in Figure 3.7. As the reader can see, it contains less edges than the one generated naturally, but more than the transitively reduced one, and preserves the meaning of the expressions belonging to `Expr`. For instance, after performing task *DP*, one can either perform task *BCO* or task *RF*, as required by expression (6).

3.5.4 Compliance with the BPMN Standard

By construction, G only contains tasks, as its nodes correspond to the tasks belonging to the expressions `Expr`. Consequently, it may be the case that some tasks $t \in V$ have several children or several parents. However, this is a bad practice in BPMN, where the control-flow must be explicit. Thus, exclusive split/merge gateways are inserted to the graph, in order to control its execution flow. As G is now really close to a BPMN process (it only lacks initial/end events), it is enriched with such events to make it a real BPMN process. The initial event is placed before the *initial nodes* of G, and the end event(s) after its *end nodes*. Depending on G's structure, the initial nodes are either the nodes of G having no parent nodes, or, if no such nodes exist, the nodes of G being the furthest from the end nodes, or, if G has no end nodes, the first node of the first expression returned by GPT.

Algorithm 1 Algorithm for Extracting Strong Sequential Constraints

Inputs: S_T (Set of Arborescences Corresponding to `Expr`)
Output: $S_<$ (Set of Strong Sequential Constraints)

$S_< \leftarrow \emptyset$
for $T_\preceq \in S_T$ **do**
    ManageTree(`root`$(T_\preceq), S_<$)
end for

procedure ManageTree(v, $S_<$)
    if $\theta(v) =$ '$<$' **then** ▷ Find sequential nodes
        for $i \in [0...|$`childs`$(v)| - 1[$ **do**
            $v_i \leftarrow$ `childs`$(v)[i]$; $v_{i+1} \leftarrow$ `childs`$(v)[i+1]$
            $\overrightarrow{T}$, $\overleftarrow{T} \leftarrow \emptyset$
            getSidemostTasksOf($T_\preceq(v_i)$, $\rightarrow$, $\overrightarrow{T}$)
            getSidemostTasksOf($T_\preceq(v_{i+1})$, $\leftarrow$, $\overleftarrow{T}$)

            for $\overrightarrow{t} \in \overrightarrow{T}$ **do** ▷ Iterate over i^{th} child's rightmost tasks
                for $\overleftarrow{t} \in \overleftarrow{T}$ **do** ▷ Iterate over $i+1^{th}$ child's leftmost tasks
                    $S_< \leftarrow S_< \cup \{(\overrightarrow{t} < \overleftarrow{t})\}$ ▷ Add strong sequential constraint
                end for
            end for
        end for
    end if

    for $v_c \in$ `childs`(v) **do**
        ManageTree(v_c, $S_<$)
    end for
end procedure

procedure getSidemostTasksOf(v, *side*, T)
    if $\theta(v) =$ `t` **then**
        $T \leftarrow T \cup \{v\}$
    else if $\theta(v) =$ '$<$' **then**
        if *side* $= \rightarrow$ **then** ▷ Recursive call on the rightmost child
            getSidemostTasksOf(`childs`$(v)[|$`childs`$(v) - 1|]$, *side*, T)
        else ▷ Recursive call on the leftmost child
            getSidemostTasksOf(`childs`$(v)[0]$, *side*, T)
        end if
    else
        for $v_c \in$ `childs`(v) **do**
            getSidemostTasksOf(v_c, *side*, T)
        end for
    end if
end procedure

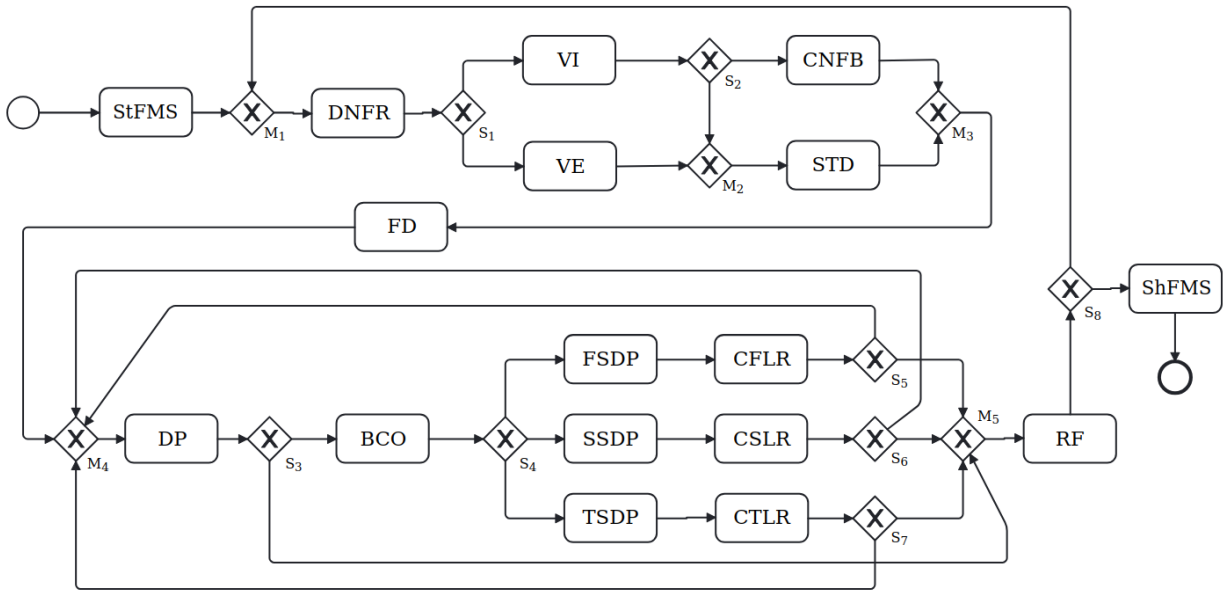


Figure 3.8: BPMN Process Obtained After Adding Exclusive Gateways and Start/End Events to the Graph Shown in Figure 3.7

Definition 3.9 (Initial Nodes)**.** *Let* $G = (V, E, \Sigma)$ *be a graph,* T *be its set of end nodes, and* `Expr` *be a list of expressions. The* initial nodes *of* G *are a set of nodes* $I \subseteq V$ *such that:*

- $I = \{v \in V \mid \texttt{parents}(v) = \emptyset\}$, *if* $|I| \neq 0$, *else*
- $I = \{v \in V \mid \forall v' \in V, \sum_{t \in T} \|(v', t)\| \leq \sum_{t \in T} \|(v, t)\|\}$, *if* $|I| \neq 0$, *else*
- $I = \{v \in V \mid \sigma(v) = \sigma(\texttt{tasks}(\texttt{Expr}[0])[0])\}$, *otherwise.*

Similarly, the end nodes of G are either the nodes of G having no children nodes, or, if no such nodes exist, the nodes of G being the furthest from the initial nodes.

Definition 3.10 (End Nodes)**.** *Let* $G = (V, E, \Sigma)$ *be a graph, and* I *be its set of initial nodes. The* end nodes *of* G *are a set of nodes*

$$T \stackrel{\text{def}}{=} \{v \in V \mid \texttt{childs}(v) = \emptyset\} \cup \{v \in V \mid c_1(v) \wedge c_2(v)\}$$

where

$$c_1(v) \Leftrightarrow \nexists v' \in V, \ \texttt{childs}(v') = \emptyset, \ s.t. \ v \rightarrow^* v'$$

and

$$c_2(v) \Leftrightarrow \forall v' \in V, \ \sum_{i \in I} \|(i, v')\| \leq \sum_{i \in I} \|(i, v)\|$$

Once done, the graph G is fully compliant with the BPMN standard (i.e., it is a valid BPMN process). However, it does not handle all the expressions of `Expr` yet.

Example. Let us consider the expressions of the running example. The BPMN process $G = (V, E, \Sigma)$ obtained after converting the sequential constraints $e_< \in \texttt{Expr}$ to a graph, and inserting exclusive gateways and start/end events to it is shown in Figure 3.8. As an example, one can see that task *CNFB* is a successor of task *VI*, meaning that task *CNFB* will be executed after task *VI*, as required by expression (2).

3.6 Management of Mutual Exclusions

The BPMN process G contains all the sequential constraints relating the tasks of the process-to-be. The next step of the generation consists in handling the *mutual exclusions* that may have been requested by the user. A *mutual exclusion* or *choice* between two tasks represents the impossibility for two tasks to be executed in the same run of a process. On a graph-based representation of a BPMN process, it can be characterised by the absence of path containing such two tasks. This notion, being the strongest form of mutual exclusion, is called *strong mutual exclusion.*

Definition 3.11 (Strong Mutual Exclusion)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process, and $\mathcal{P}^0_G$ be its set of paths. Two tasks $t_1, t_2 \in V$ are said to be* strongly mutually exclusive *if and only if there does not exist $p \in \mathcal{P}^0_G$ such that $t_1 \in p$ and $t_2 \in p$. This is written $t_1 \ |_S \ t_2$.*

Although matching exactly the spirit of the mutual exclusion, this definition is quite restrictive, in particular when dealing with cyclic BPMN processes. Indeed, cyclic graphs necessarily induce cyclic paths. In its shortest form, a cyclic path $p_{\bigcirc} \in \mathcal{P}^0_G$ can be written $(v_{B_1}, ..., v_{B_m}, v_{N_1}, ..., v_{N_n}, v_{O_1}, ...v_{O_p}, v_{N_1}, ..., v_{N_n}, v_{A_1}, ..., v_{A_q})$ where:

- the $(v_{B_1}, ..., v_{B_m})$ are the nodes reached *before* the cycle;
- the $(v_{N_1}, ..., v_{N_n})$ are the *necessarily* executed nodes of the cycle;
- the $(v_{O_1}, ...v_{O_p})$ are the *optional* nodes of the cycle;
- the $(v_{A_1}, ..., v_{A_q})$ are the nodes reached *after* the cycle.

From this decomposition, one can see that, after executing v_{N_n}, either v_{O_1} or v_{A_1} is executed, which symbolises a *choice* between these two nodes. However, both of them belong to $p_{\bigcirc}$, thus, by definition, they are not strongly mutually exclusive. To permit such behaviours, we introduce a more permissive form of mutual exclusion based on the *set of acyclic paths* of G, called *weak mutual exclusion.* An *acyclic path* is either a path of G that does not contain any repetition of its nodes, or a cyclic path of G that has been truncated so that it contains no repetition of its nodes.

Definition 3.12 (Acyclic Paths Set)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process, and let $\mathcal{P}^0_G$ be its set of paths. We define the* set of acyclic paths *of G as*

$$\widehat{\mathcal{P}}^0_G \stackrel{\text{def}}{=} \{p_{\not\bigcirc} \in \mathcal{P}^0_G\} \ \cup \bigcup_{p_{\bigcirc} = (v_1, v_2, ..., v_L, ..., v_n) \in \mathcal{P}^0_G} \{p_{\bigcirc}[: \ v_L] \mid c_1(p_{\bigcirc}, v_L) \wedge c_2(p_{\bigcirc}, v_L)\}$$

where

$$c_1(p_{\bigcirc}, v_L) \Leftrightarrow \nexists i, j \in [1...\texttt{index}(v_L)],\ i \neq j,\ s.t.\ v_i = v_j$$

and

$$c_2(p_{\bigcirc}, v_L) \Leftrightarrow p_{\bigcirc}[\texttt{index}(v_L) + 1] \in p_{\bigcirc}[: v_L]$$

Definition 3.13 (Weak Mutual Exclusion)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process, and $\widehat{\mathcal{P}}^0_G$ be its set of acyclic paths. Two tasks $t_1, t_2 \in V$ are said to be* weakly mutually exclusive *if there does not exist $p \in \widehat{\mathcal{P}}^0_G$ such that $t_1 \in p$ and $t_2 \in p$. This is written $t_1 \ |_W\ t_2$.*

Even though being more permissive than its sibling, the weak mutual exclusion may still be too restrictive when dealing with constraints expressed in natural language. For instance, according to expression (6) of the running example, the tasks *BCO* and *RF* are supposed to be mutually exclusive. However, in Figure 3.8, the acyclic path (s) $\rightarrow$ *StFMS* $\rightarrow$ $\diamondsuit_{M_1}$ $\rightarrow$ *DNFR* $\rightarrow$ $\diamondsuit_{S_1}$ $\rightarrow$ *VE* $\rightarrow$ $\diamondsuit_{M_2}$ $\rightarrow$ *STD* $\rightarrow$ $\diamondsuit_{M_3}$ $\rightarrow$ *FD* $\rightarrow$ $\diamondsuit_{M_4}$ $\rightarrow$ *DP* $\rightarrow$ $\diamondsuit_{S_3}$ $\rightarrow$ *BCO* $\rightarrow$ $\diamondsuit_{S_4}$ $\rightarrow$ *SSDP* $\rightarrow$ *CSLR* $\rightarrow$ $\diamondsuit_{S_6}$ $\rightarrow$ $\diamondsuit_{M_5}$ $\rightarrow$ *RF* $\rightarrow$ $\diamondsuit_{S_8}$ $\rightarrow$ *ShFMS* $\rightarrow$ (e) contains both tasks *BCO* and *RF*, thus preventing them from being weakly mutually exclusive. Nonetheless, after having executed task *DP*, one has to make a *choice* between executing task *BCO* or executing task *RF*, which sticks exactly to the meaning of expression (6). Thus, in the context of expression (6), tasks *BCO* and *RF* can be considered mutually exclusive. This notion, being the weakest of all, is called *context-wise (weak) mutual exclusion*.

Definition 3.14 (Context-Wise (Weak) Mutual Exclusion)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process, and let $e \in \texttt{Expr}$ be any expression such that $t_1 \in \texttt{mutex}(t_2)$. Tasks t_1 and t_2 are said to be* context-wise (weakly) mutually exclusive *if and only if they are weakly mutually exclusive in the graph $G \restriction_{\texttt{tasks}(e)}$. This is written $t_i \ |_{CW}\ t_j$.*

Based on these definitions, one can now add to G the mutually exclusive tasks not already belonging to it. The tasks not already belonging to G are $\overline{V} = \bigcup_{T \in S_T} \{v \in \texttt{tasks}(T) \mid v \notin G\}$. To perform the insertion, we must consider the set of mutually exclusive tasks of each task $t \in \overline{V}$, that is $\texttt{mutex}(t)$. For brevity, it will be abbreviated M_t. Let us then consider two possible cases: either (i) the set only contains tasks that do not belong to G, i.e., $M_t \cap V = \emptyset$, or (ii) the set contains at least one task belonging to G, i.e., $M_t \cap V \neq \emptyset$.

For case (i), the solution is rather simple: t and each task of M_t are added to the graph, as children of the start event[4]. By doing so, $\mathcal{P}^0_G$ now contains $|M_t| + 1$ new paths, containing each a single task of M_t, or t. By construction, there is no path of G containing both t and a task of M_t. Thus, they are mutually exclusive, as desired. Moreover, as the tasks of M_t are not constrained with regards to the other tasks of the graph, they will end up in parallel of the rest of the graph in the final BPMN process (see Section 3.8), thus avoiding the creation of unspecified mutual exclusions.

[4]An exclusive split gateway $\diamondsuit_S$ is also added if needed.

For case (ii), the solution is slightly more complex. Let us break M_t into two sets: the set of tasks already belonging to G called $\tilde{M}_t$, and the set of tasks not belonging to G called $\overline{M_t}$. We have that $\tilde{M}_t \cup \overline{M_t} = M_t$. A simple—yet naive—way of inserting t and the tasks belonging to $\overline{M_t}$ into G would be to do just as in case (i), that is, adding them to G as children of the start event. However, unlike in case (i), t is, by definition, constrained with regards to some tasks of the graph (the $\tilde{M}_t$). Consequently, inserting these tasks as performed in case (i) would create many unspecified mutual exclusions. To avoid this, the proposed method consists in connecting t and the tasks belonging to $\overline{M_t}$ to a particular node of G while preserving the existing mutual exclusions and limiting the number of unspecified mutual exclusions. This particular node is one of the *closest inevitable common ancestors* of the tasks belonging to $\tilde{M}_t$, and of the mutually exclusive tasks of the tasks of $\overline{M_t}$ already belonging to G.

Definition 3.15 (Closest Inevitable Common Ancestors)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process. For all $v_1, ..., v_n \in V$, the* closest inevitable common ancestors *of $(v_1, ..., v_n)$ are all the nodes $v_C \in V$ such that:*

- *For all $i \in [1...n]$, for all $\widehat{p} \in \widehat{\mathcal{P}}^0_G$, $v_i \in \widehat{p} \Rightarrow (v_C \in \widehat{p} \ \wedge \ \texttt{index}(v_C) < \texttt{index}(v_i))$ (inevitability, commonality, ancestrality);*
- *For all $p_{v_C} = (v_C, v_b, ..., v_m) \in \mathcal{P}_G(v_C)$, there does not exist $j \in [b...m]$ such that v_j is a common ancestor of $(v_1, ..., v_n)$ (closeness).*

Among the eventual multiple closest inevitable common ancestors, one of them is selected, and t and the tasks belonging to $\overline{M_t}$ are inserted to G as children of this ancestor[5]. As desired, task t is now mutually exclusive of the tasks of M_t.

Proposition 3.2 (Validity of the Closest Inevitable Common Ancestors)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process, let M_t be the set of mutually exclusive tasks of a task $t \in V$[6], let $\tilde{M}_t = M_t \cap V$, let $\overline{M}_t = M_t \setminus \tilde{M}_t$, and let V_C be the set of closest inevitable common ancestors of the tasks belonging to $\tilde{M}_t \cup \bigcup_{\bar{t} \in \overline{M_t}} \bigcup_{\substack{m \in \texttt{mutex}(\bar{t}) \\ m \in G}}$. We state that inserting t and the tasks $\overline{M}_t$ into G as children of any $v_C \in V_C$ make t weakly mutually exclusive of the tasks M_t.*

Proof. Let $G = (V, E, \Sigma)$ be a BPMN process, let M_t be the set of mutually exclusive tasks of a task $t \in V$[7], let $\tilde{M}_t = M_t \cap V$, let $\overline{M}_t = M_t \setminus \tilde{M}_t$, and let V_C be the set of closest inevitable common ancestors of the tasks belonging to $\tilde{M}_t \cup \bigcup_{\bar{t} \in \overline{M_t}} \bigcup_{\substack{m \in \texttt{mutex}(\bar{t}) \\ m \in G}}$. We will show that, for all $v_C \in V_C$, adding t and the tasks of $\overline{M}_t$ as children of v_c make t weakly mutually exclusive of the tasks belonging to M_t.

[5] An exclusive split gateway $\langle \times \rangle_S$ is also added if needed.

[6] One could take $t \notin V$ without changing the validity of the statement.

[7] One could take $t \notin V$ without changing the validity of the statement.

Adding t and the tasks $\overline{M_t}$ as children of v_C creates a BPMN process $G' = (V', E', \Sigma')$, where:

— $V' = V \cup \overline{M_t} \cup \{t\}$;
— $E' = E \cup \bigcup_{\bar{t} \in \overline{M_t}} \{v_C \to \bar{t}\} \cup \{v_C \to t\}$;
— $\Sigma' = \Sigma \cup \bigcup_{\bar{t} \in \overline{M_t}} \{\sigma(\bar{t})\} \cup \{\sigma(t)\}$.

Consequently, we have that

$$\widehat{\mathcal{P}}^0_{G'} = \widehat{\mathcal{P}}^0_G \cup \bigcup_{\bar{t} \in \overline{M_t}} \bigcup_{p \in \widehat{\mathcal{P}}^0_G} \{(p[: v_C], \bar{t}) \mid v_C \in p\} \cup \bigcup_{p \in \widehat{\mathcal{P}}^0_G} \{(p[: v_C], t) \mid v_C \in p\}$$

By construction, there is no $p \in \widehat{\mathcal{P}}^0_{G'}$ containing both t and a task $\bar{t} \in \overline{M_t}$. Moreover, by definition of v_C, we have that for all $\tilde{t} \in \tilde{M_t}$, for all $p \in \widehat{\mathcal{P}}^0_{G'}$, $\tilde{t} \in p \Rightarrow p = (v_1, ..., v_C, ..., \tilde{t}, ..., v_z)$. Thus, by construction of G', there is no $p \in \widehat{\mathcal{P}}^0_{G'}$ containing both t and a task $\tilde{t} \in \tilde{M_t}$. Consequently, t is weakly mutually exclusive of all $\tilde{t} \in \tilde{M_t}$, and of all $\bar{t} \in \overline{M_t}$, which corresponds to all the tasks of M_t. □

Remark 3.2. *It is worth mentioning that considering the closest inevitable common ancestor of (the barbarian expression)* $\tilde{M_t} \cup \bigcup_{\bar{t} \in \overline{M_t}} \bigcup_{\substack{m \in \texttt{mutex}(\bar{t}) \\ m \in G}}$ *is mandatory in order to preserve the mutual exclusions stated in* `Expr`*. Indeed, adding the tasks of* $\overline{M_t}$ *as children of the closest inevitable common ancestor of* $\tilde{M_t}$ *only could potentially prevent a task* $\bar{t} \in \overline{M_t}$ *from being mutually exclusive of one of its mutually exclusive tasks, in the case where such a task is a predecessor of the closest inevitable common ancestor of* $\tilde{M_t}$*.*

These new connections, although having the benefit of adding the desired mutual exclusions to G, may have generated unspecified mutual exclusions. Indeed, t and the tasks $\bar{t} \in \overline{M_t}$ are connected only to the closest inevitable common ancestor of $\tilde{M_t} \cup \bigcup_{\bar{t} \in \overline{M_t}} \bigcup_{\substack{m \in \texttt{mutex}(\bar{t}) \\ m \in G}}$, making them mutually exclusive of a possibly large part of the graph. To reduce the number of unspecified mutual exclusions, an attempt is made to connect t and the tasks $\bar{t} \in \overline{M_t}$ to the children of the tasks belonging to $\tilde{M_t}$. When a child can be connected, it becomes a child of t, and of each task $\bar{t} \in \overline{M_t}$[8]. If a child cannot be connected without discarding a desired mutual exclusion, a new attempt is made on the children of this child, and so on until reaching the end of the graph. By doing so, the new mutual exclusions are limited to the tasks appearing between the selected closest inevitable common ancestor and the connected children. Still, some of these tasks may not belong to $\tilde{M_t}$, thus leading to unspecified—yet unavoidable—mutual exclusions. G now satisfies a new set of constraints $Cons_2$ which contains the mutual exclusions.

[8] An exclusive split gateway  is also added if needed.

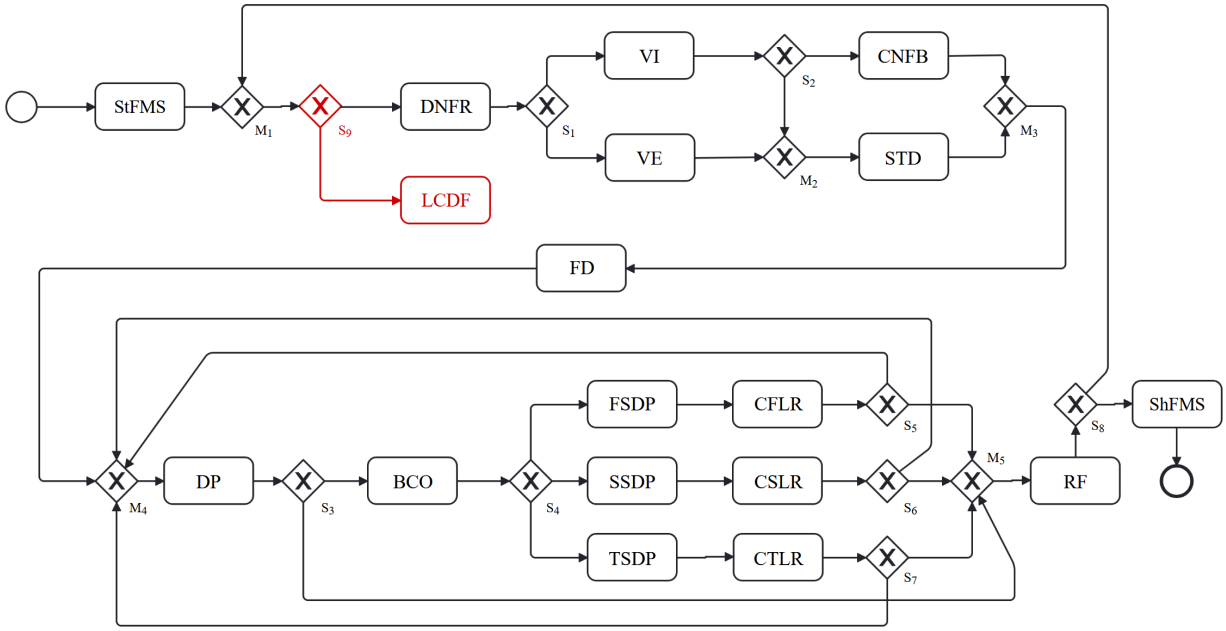


Figure 3.9: BPMN Process Resulting from the Addition of Task *LCDF* as Child of M_1

Example. Let us consider the BPMN process G in Figure 3.8. Five tasks have a set of mutually exclusive tasks containing tasks that do not appear yet in G: *DNFR*, *VI*, *VE*, *CNFB* and *STD*. Indeed, according to expression (4) of the running example, they are all mutually exclusive of task *LCDF*, which does not belong to G. To add this task to G, the first step consists in finding the closest inevitable common ancestors of the other tasks *DNFR*, *VI*, *VE*, *CNFB* and *STD*. These tasks have a single closest inevitable common ancestor: the exclusive merge gateway M_1. Task *LCDF* is then added as a child of this merge gateway, as highlighted in red in Figure 3.9[9].

The second part of the insertion consists in trying to connect *LCDF* to the children of *DNFR*, *VI*, *VE*, *CNFB* and *STD*, in order to reduce the number of unspecified mutual exclusions. The only eligible child of *DNFR*, *VI*, *VE*, *CNFB* and *STD* is the exclusive merge gateway M_3, as the others are *VI*, *VE*, *CNFB*, *STD*, and the exclusive split and merge gateways S_1, S_2 and M_2, which must be mutually exclusive of *LCDF*. *LCDF* is thus connected to M_3, as highlighted in red in Figure 3.10. As there is no path containing both *DNFR* and *LCDF*, nor *VI* and *LCDF*, nor *VE* and *LCDF*, nor *CNFB* and *LCDF*, nor *STD* and *LCDF*, M_3 is a valid child node. As there is no other eligible child node, the computation stops here, with *LCDF* having been added to G. It is worth noting that, in this case, the insertion did not add any unspecified mutual exclusion to the BPMN process.

[9]An exclusive split gateway S is added between these two nodes to preserve the BPMN semantics.

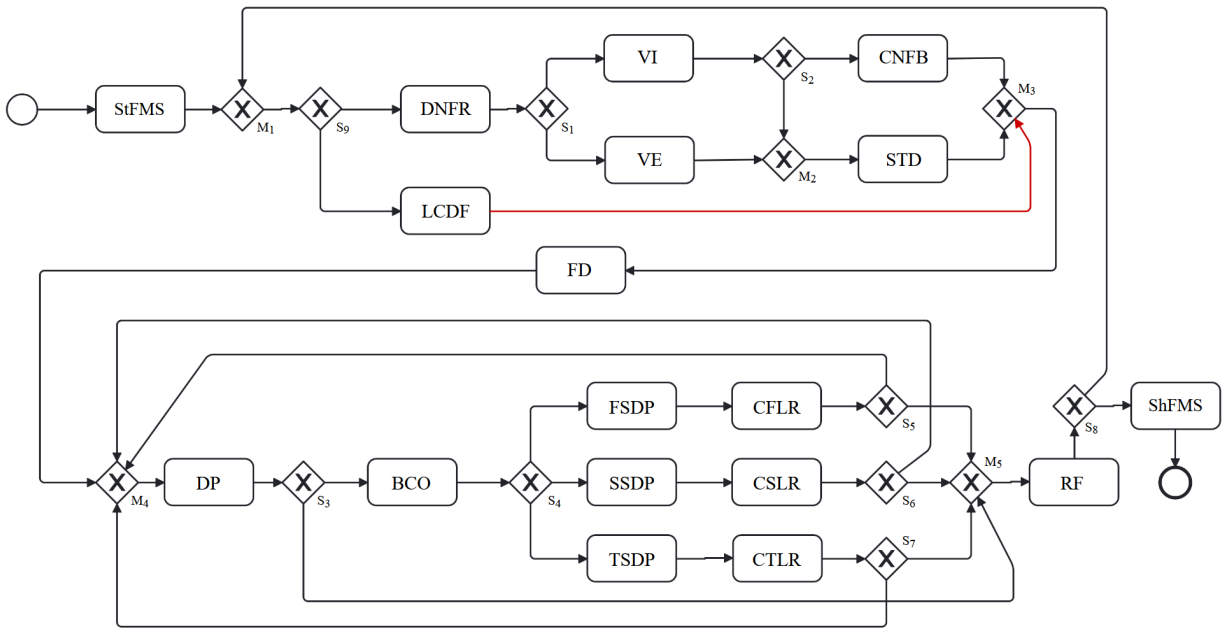


Figure 3.10: BPMN Process Resulting from the Addition of Task *LCDF* as Parent of $\times_{M_3}$

3.7 Management of Explicit Loops

The explicit loops—by opposition to the implicit ones induced by cyclic sequential constraints—are the loops belonging to the set of loops called `Loops`. To appear in G, each loop $L = (v_1, ..., v_n) \in$ `Loops` must form a strongly connected component in G. Depending on the composition of L, the insertion is performed differently. We can distinguish between two major cases: either (i) none of the nodes in L belong to G, i.e., $L \cap V = \emptyset$ or (ii) at least one node in L belongs to G, i.e., $L \cap V \neq \emptyset$. By definition, the nodes $v \in L \setminus V$ cannot be mutually exclusive of any other node, otherwise they would have been added to G in the previous step, nor sequentially constrained to any other node, otherwise they would already belong to G. Thus, they are not constrained with regards to any other node of G.

In case (i), the approach simply consists in connecting all the $v \in L$ to a parallel split gateway $+_S$ and to a parallel merge gateway $+_M$. This construct is then put inside a loop, that is, between an exclusive merge gateway $\times_M$ and an exclusive split gateway $\times_S$ which is connected to the exclusive merge gateway. Once done, the exclusive merge gateway is inserted in G as child of the initial event[10]. By doing so, the loop nodes now form a strongly connected component that belongs to G, as desired. More formally, given that (s) (resp. (e)) is an initial (resp. final) node of G, we have that $G_3 = (V_3, E_3, \Sigma_3)$,

[10] An exclusive split gateway $\times_S$ is added if needed.

where:

— $V_3 = V \cup L \cup \{$  $\} \cup \tilde{V}$;

— $E_3 = E \setminus \{$  $\} \cup \{$        $\} \cup \tilde{E}$;

— $\Sigma_3 = \bigcup_{v \in V_3} \{\sigma(v)\}$.

where

$$\tilde{V} = \begin{cases} \emptyset & \text{if } |L| = 1 \\ \{ \text{} \} & \text{otherwise} \end{cases}$$

and

$$\tilde{E} = \begin{cases} \{ \text{} \} & \text{if } |L| = 1 \\ \{ \text{  } \} \cup \bigcup_{v \in L} \{ \text{} \} & \text{otherwise} \end{cases}$$

In case (ii), some $v \in L$ already belong to G, while some others do not. However, the loop nodes belonging to G may be completely disconnected, already connected, or partially connected, thus no assumption can be made on how to connect them. If they do not form a strongly connected component yet, they have to be connected to make this strongly connected component appear in G. This starts by computing all the components of $G \restriction_{L \cap V}$.

These components represent disconnected portions of L that one has to connect to make the strongly connected component corresponding to L appear in G. However, this must be done carefully in order not to break any mutual exclusion already handled by G. Indeed, connecting two such components consists in adding new flows to G, which has an impact on its paths, and thus, potentially, on its mutual exclusions. To ensure that L will be added to G if it is possible (i.e., if it does not intrinsically break some mutual exclusions), all permutations of the components of $G \restriction_{L \cap V}$ are computed. Each such permutation represents a possible order in which the components can be connected to the others to make the loop appear in G. For each such permutation, the n^{th} component is connected to the $n + 1^{\text{th}}$ component. This is done by connecting each node of the n^{th} component having a *0-reachability* to the set of nodes of the $n + 1^{\text{th}}$ component ensuring an *∞-reachability*.

Definition 3.16 (n-reachability of a Node)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process. For all $v \in V$, the* n-reachability *of v is the number of nodes that v can reach, i.e.,*

$$n = |\{v' \in V \setminus \{v\} \mid v \rightarrow^* v'\}|$$

By convention, if $n = |V| - 1$ (i.e., if v can reach all the nodes of G), v is said to have an ∞-reachability.

This notion can be extended to a set of nodes.

Definition 3.17 (n-reachability of a Set of Nodes)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process. For all $\{v_1, ..., v_m\} \subseteq V$, the* n-reachability *of $\{v_1, ..., v_m\}$ is the number of nodes that $\{v_1, ..., v_m\}$ can reach, i.e.,*

$$n = |\bigcup_{v_i \in \{v_1,...,v_m\}} \{v' \in V \setminus \{v_1, ..., v_m\} \mid v_i \rightarrow^* v'\}|$$

By convention, if $n = |V| - m$ (i.e., if the $\{v_1, ..., v_m\}$ can reach all the nodes of G), the set $\{v_1, ..., v_m\}$ is said to have an ∞-reachability.

Such a connection ensures that each node of the n^{th} component can now reach every node of the $n + 1^{\text{th}}$ component, and also that the n^{th} and $n + 1^{\text{th}}$ components now form a single component. If during the connection phase, the n^{th} component of a permutation cannot be connected to the $n + 1^{\text{th}}$ component without breaking some existing mutual exclusions, the permutation is discarded. Once a valid permutation is found, the remaining ones are discarded. If no valid permutation is found, the explicit loop L is not added to G. Finally, if some tasks of the loop did not already belong to G, they are arbitrarily added between two connected components, using the same method than in case (i). This step results in G satisfying another set of constraints $Cons_3$.

Proposition 3.3 (Validity of the Components Connection)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process, let $L = (v_1, ..., v_n) \in$ `Loops` be a loop that should be added to G, and let $\{G_1, ..., G_m\}$ be the set of components of $G \restriction_{L \cap V}$. We state that for all $i \in [1...m - 1]$, connecting all the $\{v_1, ..., v_o\} \in G_i$ having a 0-reachability to the (smallest) set of $\{v_1, ..., v_p\} \in G_{i+1}$ ensuring an ∞-rechability, and all the $\{v_1, ..., v_q\} \in G_m$ having a 0-reachability to the (smallest) set of $\{v_1, ..., v_r\} \in G_1$ ensuring an ∞-rechability make L become a strongly connected component in G.*

Proof. Let $G = (V, E, \Sigma)$ be a BPMN process, let $L = (v_1, ..., v_n) \in$ `Loops` be a loop that should be added to G, and let $\{G_1, ..., G_m\}$ be the set of components of $G \restriction_{L \cap V}$. Let us separate the proof into two parts.

First, let us show that connecting all the nodes of a component having a 0-reachability to the (smallest) set of nodes having an ∞-reachability makes the component become a strongly connected component. Let $\{v_a, ..., v_m\}$ be the set of nodes of G_1 having a 0-reachability, and let $\{v_n, ..., v_z\}$ be its (smallest) set of nodes ensuring an ∞-reachability. Adding an edge connecting each $v_i \in \{v_a, ..., v_m\}$ to each $v_j \in \{v_n, ..., v_z\}$ ensures that each $v_i \in \{v_a, ..., v_m\}$ now has a ∞-reachability. Moreover, by definition, each $v_k \notin \{v_a, ..., v_m\} \cup \{v_n, ..., v_z\}$ must have at least a 1-reachability (otherwise it would belong to the $\{v_a, ..., v_m\}$). Hence, it must be able to reach at least one $v_i \in \{v_a, ..., v_m\}$. However, we know that each $v_i \in \{v_a, ..., v_m\}$ now has an ∞-reachability. Thus, each $v_k \notin \{v_a, ..., v_m\} \cup \{v_n, ..., v_z\}$ now has an ∞-reachability. As each $v \in V$ now has a an ∞-reachability, G is a strongly connected component.

Then, let us show that connecting two components G_1 and G_2 with the same method

creates another component $G_{1,2}$. Let $\{v_{1_a}, ..., v_{1_m}\}$ be the set of nodes of G_1 having a 0-reachability, let $\{v_{1_n}, ..., v_{1_z}\}$ be its (smallest) set of nodes ensuring an ∞-reachability, let $\{v_{2_a}, ..., v_{2_m}\}$ be the set of nodes of G_2 having a 0-reachability, and let $\{v_{2_n}, ..., v_{2_z}\}$ be its (smallest) set of nodes ensuring an ∞-reachability. Adding an edge connecting each $v_i \in \{v_{1_a}, ..., v_{1_m}\}$ to each $v_l \in \{v_{2_n}, ..., v_{2_z}\}$ ensures that $G_{1,2} = (V_1 \cup V_2, E_1 \cup E_2 \cup \{v_i \rightarrow v_l \mid v_i \in \{v_{1_a}, ..., v_{1_m}\} \ \wedge \ v_l \in \{v_{2_n}, ..., v_{2_z}\}\}, \Sigma_1 \cup \Sigma_2)$ is a component. Moreover, it also ensures that the $\{v_{1_n}, ..., v_{1_z}\}$ have an ∞-reachability in this component. By construction, we also have that the $\{v_{2_a}, ..., v_{2_m}\}$ still have a 0-reachability. Thus, we created a component $G_{1,2}$ that contains both G_1 and G_2, and which has the same (smallest) set of ∞-reachability nodes than G_1, and the same set of 0-reachability nodes than G_2.

We showed that for all $i \in [1...m-1]$, connecting each G_i to each G_{i+1} creates a component G_f containing all the G_i, and whose (smallest) set of ∞-reachability nodes is the same than G_1, while its set of 0-reachability nodes is the same than G_m. Finally, we showed that connecting this set of 0-reachability nodes to the set of ∞-reachability nodes makes G_f a strongly connected component, which is our goal. □

Example. Let us consider the explicit loops of the running example `Loops`. This set contains a single explicit loop l, containing tasks *FSDP*, *SSDP*, *TSDP*, *CFLR*, *CSLR* and *CTLR*, as described by expression (9). To manage this loop, the first step consists in computing the components of $G \restriction_l$. $G \restriction_l$ has three components: G_1, containing tasks *FSDP* and *CFLR*, G_2, containing tasks *SSDP* and *CSLR*, and G_3, containing tasks *TSDP* and *CTLR*. To make l appear in G, these three components must be connected. The possible connection orderings are the following: $\{G_1, G_2, G_3\}$, $\{G_1, G_3, G_2\}$, $\{G_2, G_1, G_3\}$, $\{G_2, G_3, G_1\}$, $\{G_3, G_1, G_2\}$ and $\{G_3, G_2, G_1\}$. The first permutation means that G_1 must be connected to G_2, G_2 must be connected to G_3, and G_3 must be connected to G_1. To do so, the nodes of G_1 having a 0-reachability must be connected to the set of nodes of G_2 ensuring an ∞-reachability. Here, this means that *CFLR* must be connected to *SSDP*. By doing so, *FSDP* and *SSDP* are no longer weakly mutually exclusive. However, they remain context-wise mutually exclusive, thus the connection is valid. Similarly, connecting G_2 and G_3 consists in connecting *CSLR* to *TSDP*, which also breaks the weak mutual exclusion between *SSDP* and *TSDP*, but not their context-wise mutual exclusion. Finally, G_3 and G_1 are connected by adding an edge between *CTLR* and *FSDP*, which also preserves the context-wise mutual exclusion between *FSDP* and *TSDP*. As the permutation is valid, the remaining ones are discarded. Figure 3.11 shows the resulting graph, in which tasks *FSDP*, *SSDP*, *TSDP*, *CFLR*, *CSLR* and *CTLR* are now in a loop containing only them[11].

3.8 Management of Parallelism

At this stage of the BPMN process generation, two operators of the language are still to be managed: the ',' and the '&' operators. The ',' operator separates tasks that are

[11] To preserve the BPMN semantics, three exclusive merge gateways M_6, M_7, and M_8 were added to G.

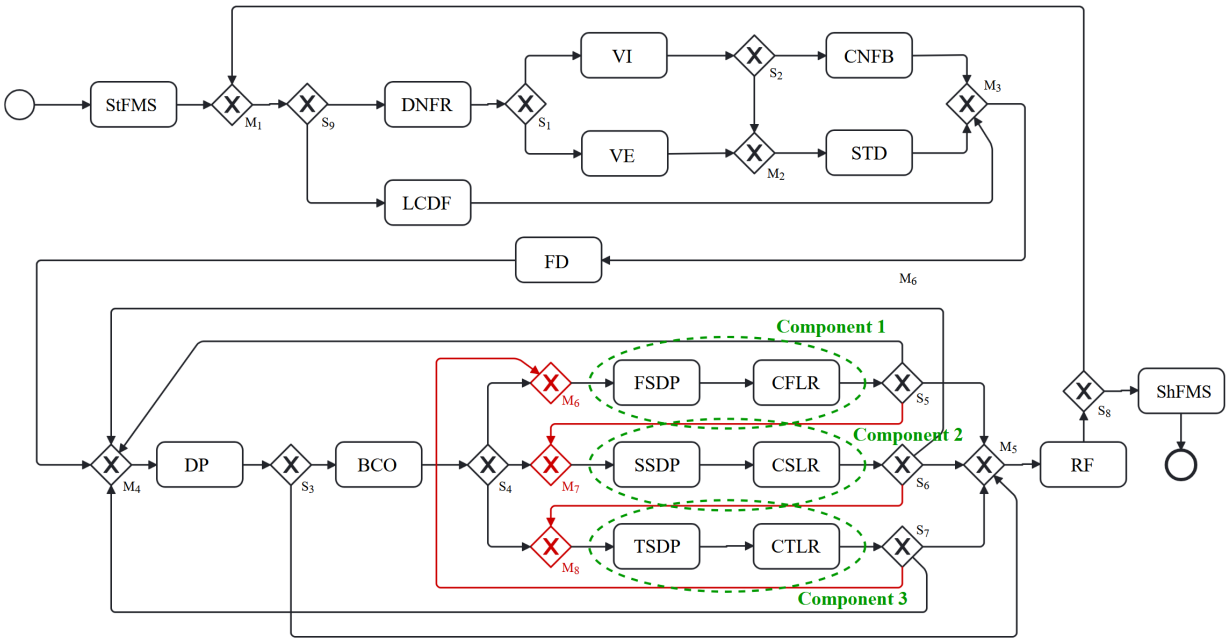


Figure 3.11: BPMN Process Resulting from the Addition of the Loop $(FSDP, SSDP, TSDP, CFLR, CSLR, CTLR)*$

not constrained to each others, while the '&' operator separates tasks that should be put in parallel. However, in BPMN, parallelism precisely describes the absence of constraints between elements, as two parallel elements may end up executing sequentially in any order, or at the same time. Thus, we decided to manage the ',' and the '&' operators the same way.

In this approach, adding parallelism to the graph consists in replacing the exclusive split (resp. merge) gateways by parallel split (resp. merge) gateways. However, replacing an exclusive gateway by a parallel gateway may lead to potentially severe issues if not done carefully. In BPMN, parallelism can induce two major issues: *deadlocks* and *livelocks*. A deadlock occurs whenever a parallel merge gateway does not (and will not) receive a sufficient number of tokens to be triggered, that is, one token per incoming flow. This phenomenon prevents the gateway from merging its incoming tokens, and thus from sending a token to its outgoing flow. Consequently, the process cannot complete its execution. A livelock occurs whenever a parallel split gateway can be reached infinitely often by a token that it sent and which was not merged with its siblings before reaching this parallel split again. This parallel split thus produces new tokens infinitely often, thus preventing the process from terminating. Such behaviour happens when there is no node *synchronising all the paths* starting from the children of a parallel split gateway that can reach itself.

Definition 3.18 (Paths Synchronisation Node)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process, and* $\mathcal{P}_G$ *be its corresponding set of paths. A node* $v \in V$ *is said to* synchronise a set of paths $\mathcal{P}'_G \subseteq \mathcal{P}_G$ *if and only if for all* $p \in \mathcal{P}'_G$, $v \in p$. $\mathcal{P}'_G$ *may have several synchronisation*

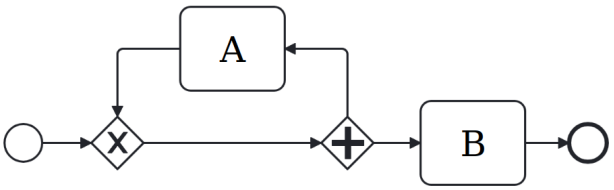


Figure 3.12: Basic Example of (Syntactic) Livelock in BPMN

nodes, which can be computed using the $\texttt{sync}(\mathcal{P}'_G)$ *operator:*

$$\texttt{sync}(\mathcal{P}'_G) = \{v \in V \mid \forall p \in \mathcal{P}'_G,\ v \in p\}$$

Definition 3.19 ((Syntactic) Livelock)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process, and let* $v \in V$ *be a parallel split gateway.* G *contains a* (syntactic) livelock *if:*

- $\exists v_c \in \texttt{childs}(v)$ *such that* $v_c \rightarrow^* v$ *(*v_c *can reach* v*);*
- $\nexists v_s \in V$ *such that:*
  - $v_s \in \texttt{sync}(\bigcup_{v_c \in \texttt{childs}(v)} \mathcal{P}_G(v_c))$*;*
  - $\forall p = (v_1, ..., v_s, ..., v_n) \in \bigcup_{v_c \in \texttt{childs}(v)} \mathcal{P}_G(v_c),\ \exists i \in [1...n] \mid v_i = v \Rightarrow \texttt{index}(v_s) < i.$

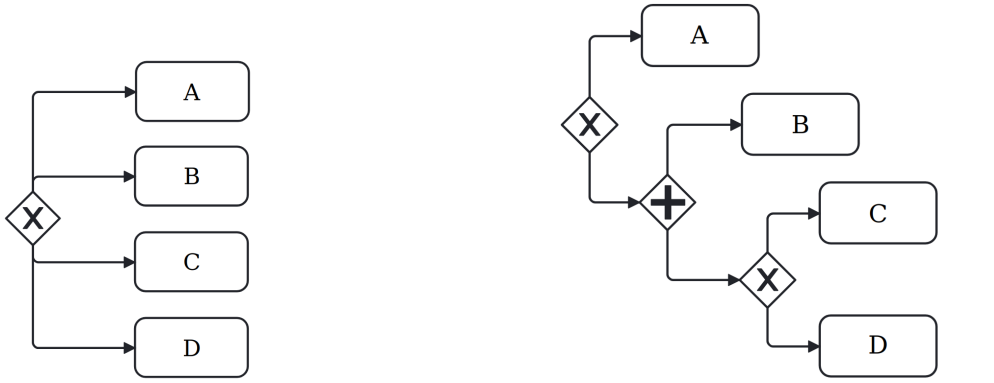
(either the paths starting from children of v *are not synchronised, or at least one of them can reach* v *before reaching* v_s*)*

Example. Figure 3.12 illustrates this potential issue on a simple BPMN process. As the reader can see, the parallel split gateway sends a token τ_1 to B, which reaches the end event, and another token τ_2 to A. These two tokens are never merged, and τ_2 eventually reaches the parallel split gateway. When it receives τ_2, it sends τ'_1 to B, and τ'_2 to A. As the parallel split gateway is triggered infinitely often, there is no way for this process to eventually complete its execution, as it has no possibility to prevent itself from producing new tokens. Thus, this process contains a livelock.

3.8.1 Insertion of Parallel Gateways

As mentioned earlier, inserting parallel gateways to the graph mostly consists in replacing some exclusive gateways by parallel ones while avoiding syntactic livelocks. However, simply switching the type of a gateway is often not sufficient to handle properly all the possible forms that the process could take. Indeed, a simple exclusive split gateway with n children tasks could generate almost $2 \times (2^n - 1)$ syntactically different BPMN constructs.

Example. Let us consider the four mutual exclusion constraints $A \mid B$, $A \mid C$, $A \mid D$, and $C \mid D$. Before starting to insert parallelism in it, the BPMN subprocess corresponding to these four constraints would be the one shown in Figure 3.13(a). However, if one wants to parallelise the tasks that can be parallelised without breaking any of the four



(a) Original BPMN Sub-Process (b) Most Parallel BPMN Sub-Process

Figure 3.13: Example of Parallel Tasks Combination

aforementioned mutual exclusions, while adding as much parallelism as (s)he can, (s)he would build the BPMN sub-process displayed in Figure 3.13(b). It is as parallelised as possible, as the only non-parallel tasks are the ones that should be mutually exclusive. For instance, according to the given constraints, tasks B and C are not mutually exclusive, although they are in Figure 3.13(a). However, after the insertion of parallel gateways, they end up in parallel, as they should be.

Inserting Parallel Splits

The generation of parallel structures, such as the one presented above, relies on combinatorics to build all the possible such structures given a set of tasks. This is ensured by a function $\eta : V^n \to W$, assuming that W is the set of all existing workflows. Function η is recursively defined and its return value is, for clarity, written in the form of expressions compliant with the language defined in Section 3.2.

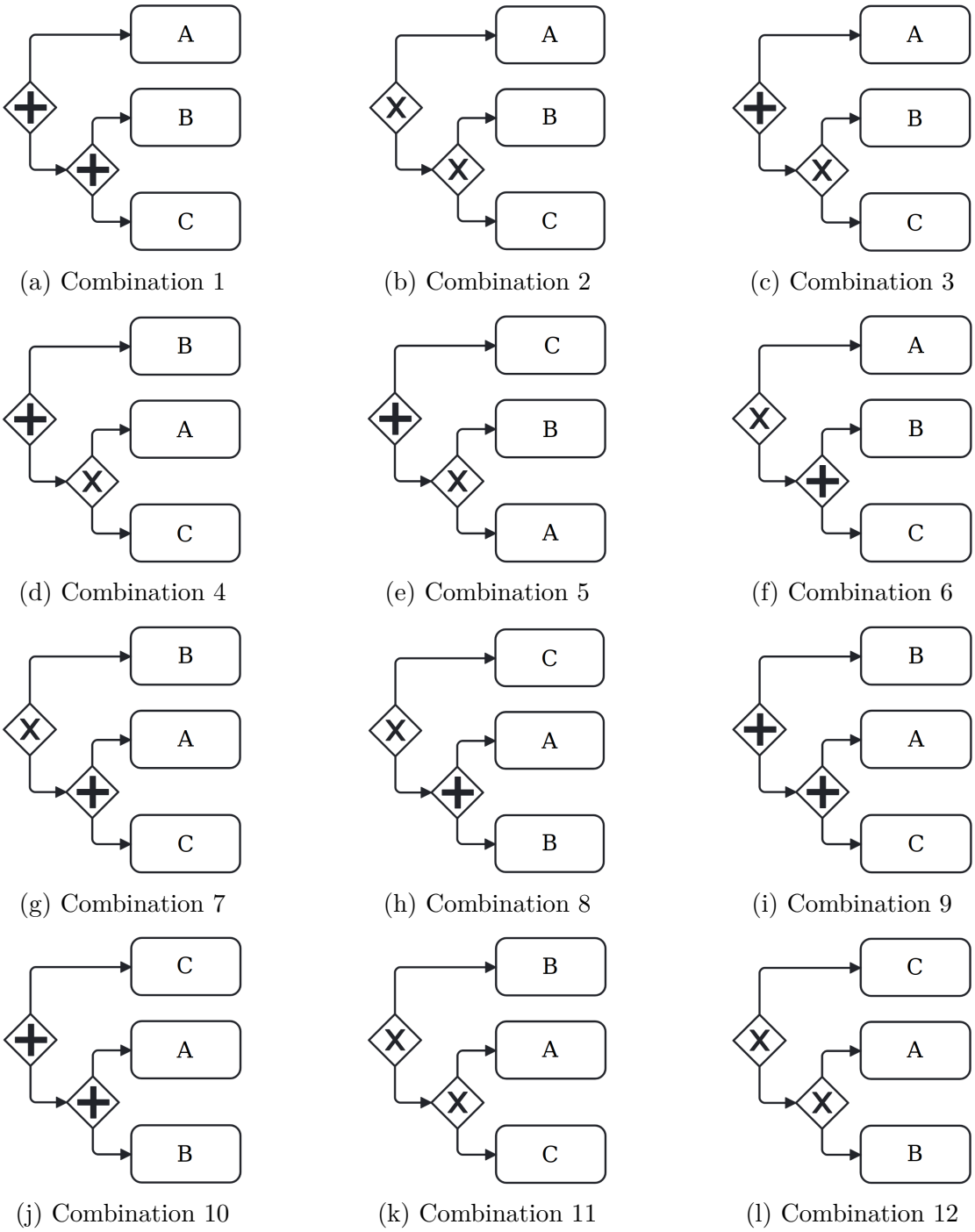
Definition 3.20 (Generation of Parallel Split Structures)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process, and let $g \in V$ be an exclusive split gateway having n children tasks $(t_1, ..., t_n) \in V$ whose parallel structures have to be generated. Function η is (partially) defined as:*

$$\eta(\{t_1, ..., t_n\}) = \begin{cases} \{t_1\} & \text{if } |\{t_1, ..., t_n\}| = 1 \\ \{\eta(\widehat{\{t_1, ..., t_n\}}) \mid \eta(\overline{\{t_1, ..., t_n\}})\} \cup \{\eta(\widehat{\{t_1, ..., t_n\}}) \ \& \ \eta(\overline{\{t_1, ..., t_n\}})\} & \text{else} \end{cases}$$

where $\widehat{\{t_1, ..., t_n\}} = \texttt{any}(2_{1^+}^{\{t_1,...,t_n\}}, 1)$ *and* $\overline{\{t_1, ..., t_n\}} = \{t_1, ..., t_n\} \setminus \widehat{\{t_1, ..., t_n\}}$.

Example. Given 3 tasks A, B, and C, the η function generates 12 syntactically different BPMN subprocesses, that are presented in Figure 3.14.

Applying this operation generates several syntactically different graphs. These graphs are built in a BPMN-like fashion, which allows the appliance of the minimisation rules



(a) Combination 1 (b) Combination 2 (c) Combination 3

(d) Combination 4 (e) Combination 5 (f) Combination 6

(g) Combination 7 (h) Combination 8 (i) Combination 9

(j) Combination 10 (k) Combination 11 (l) Combination 12

Figure 3.14: Example of Parallel Tasks Combination

described in Definition 2.24. The application of these rules returns a set of graphs that are now semantically different. For each of them, a copy of G is made, in which the exclusive split from which these graphs were produced is replaced by the current generated graph. Among all these copies, only the ones not creating any syntactic livelock, nor violating any desired mutual exclusion, are kept, while the others are discarded. They are called *syntactically compliant processes*.

Definition 3.21 (Syntactically Compliant BPMN Process)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process.* G *is said to be* syntactically compliant *if for all* $v \in V$ *such that* $\theta(v) = \diamondsuit_S$, v *is* syntactically compliant*.*

Remark 3.3. *A BPMN process containing no parallel split gateway is trivially considered as syntactically compliant.*

Definition 3.22 (Syntactically Compliant Parallel Split Gateway)**.** *Let* $G = (V, E, \Sigma)$ *be a graph. For all* $v \in V$ *such that* $\theta(v) = \diamondsuit_S$, v *is said to be* syntactically compliant *if and only if:*

- v *does not create any* syntactic livelock *(i.e., it complies with Definition 3.19);*
- v *does not break any* desired mutual exclusion*, i.e.,* $\forall v_1, v_2 \in \texttt{childs}(v),\ v_1 \neq v_2$:

  $(\forall t_1 \in T_1,\ \nexists t_2 \in T_2 \mid t_2 \in \texttt{mutex}(t_1)) \ \wedge\ (\forall t_2 \in T_2,\ \nexists t_1 \in T_1 \mid t_1 \in \texttt{mutex}(t_2))$

  where $\forall i \in \{1, 2\},\ T_i = \bigcup_{p \in \mathcal{P}_G(v_i)} \texttt{tasks}(p[: \texttt{sync}(\mathcal{P}_G(v_1) \cup \mathcal{P}_G(v_2))])$

Inserting Parallel Merges

Each copy of G now contains its parallel split gateways, but no parallel merge gateways yet. The insertion of these parallel merge gateways, essential to prevent deadlocks and livelocks in the process, is done in two sequential steps. First, the parallel split gateways previously added are analysed to check whether they require a synchronisation node to avoid creating syntactic livelocks in the process. If this is the case, a parallel merge gateway is inserted before this synchronisation node, and becomes the new synchronisation node of the parallel split. This ensures that the parallel split gateway will not create syntactic livelocks in the final process. Next, the remaining exclusive merge gateways of the BPMN process are checked to see whether their closest common ancestor is a parallel split. If this is the case, a parallel version of this gateway may not suffer from deadlocks, so it is switched to a parallel gateway.

3.8.2 Detection of Deadlocks/Livelocks and Parallelism Removal

Our previous modifications of the copies of G introduced parallelism in them. Although performing this parallelisation phase carefully, it is rather complex to ensure the absence of deadlocks or livelocks at design time by performing only a syntactic analysis of the process. However, such behaviours can be easily detected when *executing* the BPMN process.

Definition 3.23 ((Execution) Deadlock)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process.* G *contains* a(n execution) deadlock *whenever there exists* $C \in \mathcal{H}(G)$ *such that:*

— $C = \texttt{push}(C)$*;*
— C *is not a* final configuration*.*

Definition 3.24 ((Execution) Livelock)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process.* G *contains* a(n execution) livelock *whenever there exist* $C, C' \in \mathcal{H}(G)$ *such that:*

— $\forall n \in C,\ C[n] = 0 \Leftrightarrow C'[n] = 0$*;*
— $\forall n \in C,\ C'[n] \geq C[n]$*;*
— $\exists n \in C$ *such that* $C'[n] > C[n]$*.*

Detection of Deadlocks/Livelocks

The detection of such configurations is based on simulation of the given BPMN process. However, the simulation that we use to detect such erroneous configurations slightly differs from the one presented in Section 2.4.3. Indeed, in Section 2.4.3, we perform one simulation of the process, representing one of its possible executions. Here, we want to ensure that there is no possible execution of the process that reaches a deadlock or a livelock. In our context, there is a single type of node possibly making the execution of a process differ from another: the exclusive split gateway. Indeed, when a token reaches this node, it is sent to any of its children nodes, non-deterministically. To ensure that the detection takes into account all the possible configurations of the process, the solution that we opted for consists in duplicating the current configuration every time a token must be sent away from an exclusive split gateway. The simulation no longer returns a single history $\mathcal{H}$, but a set of histories $S_{\mathcal{H}} = \{\mathcal{H}_1, ..., \mathcal{H}_n\}$, each of which corresponds to a possible execution of the process. However, this modification introduces a major issue: a simulation may no longer terminate. Originally, simulating a correct process (i.e., without deadlock/livelock) necessarily terminates, due to the semantics of the different BPMN operators, and to the fact that the children of an exclusive split gateway are probabilistically chosen. From now on, as all the children of an exclusive split gateway receive a token from their parent, certain configurations may remain in a state where, for instance, a token is continuously circulating through a strongly connected component of the process. To avoid such situations, the simulation now makes use of a *fixed point* analysis to terminate.

Definition 3.25 (Fixed Point)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process.* $\mathcal{H}(G)$ *contains a* fixed point *whenever there exist* $C, C' \in \mathcal{H}(G)$ *such that:*

— $\forall n \in C,\ C[n] = C'[n]$*;*
— $\forall n' \in C',\ C[n'] = C'[n']$*.*

When a fixed point is reached, the simulation stops generating all the possible configurations, and is asked to terminate. This is ensured by a mechanism that forces the simulator to transmit the tokens of an exclusive split gateway only to its child that is the closest to an end event. When the simulation has terminated, the set of histories $S_{\mathcal{H}}$ is analysed

to verify whether there exists an history $\mathcal{H} \in S_{\mathcal{H}}$ containing a deadlock or a livelock. If not, the BPMN process remains as is. Otherwise, some of its parallel elements have to be removed.

Parallelism Removal

If the deadlock/livelock detector found a deadlock or a livelock in the current BPMN process, this process has to be modified. Although being rather simple to detect deadlocks/livelocks in processes, it is more difficult to identify their sources. For instance, a node holding a token in a deadlock configuration may not be the source of that deadlock. The simplest solution that we found to identify the source of such errors so far consists in removing step by step the parallel gateways of the BPMN process, until reaching a graph containing no deadlock/livelock. The removal is made in a simple way: each parallel gateway of the process is replaced by an exclusive one, which leads to the generation of several new BPMN processes, each of them containing one less parallel gateway. If the removed gateway is a parallel merge gateway, and if this gateway is the mandatory synchronisation node of a parallel split gateway, this parallel split gateway is also replaced by an exclusive split gateway to avoid syntactic livelocks. Among all the deadlock/livelock-free generated BPMN processes, the one with the largest number of parallel tasks is elected as best candidate. This procedure is summarised in Algorithm 2. This final BPMN process now satisfies a new set of constraints $Cons_4$.

Example. Let us consider the current version of the BPMN process, that is, the one shown in Figure 3.11. It contains 9 exclusive split gateways, that are candidates for being replaced by parallel split gateways. Among these exclusive split gateways, only two are valid candidates: S_1 and S_2. Indeed, the 7 remaining gateways would all generate an incorrect behaviour if switched to parallel:

— Gateway S_3 would not satisfy expression (6) of the running example anymore;
— Gateways S_4, S_5, S_6, S_7, and S_8 would create a syntactic livelock;
— Gateway S_9 would break the mutual exclusion between, for instance, task *DNFR* and task *LCDF* (expression (4)).

Thus, S_1 and S_2 become parallel split gateways. To avoid any syntactic livelock, M_3 must become a parallel merge gateway. However, simply switching the type of M_3 to parallel would create a deadlock. Indeed, as the gateway S_9 is an exclusive split gateway, M_3 would receive a token either from task *DNFR*, or from task *LCDF*, but not from both, thus M_3 would never be able to merge its incoming tokens, preventing it from sending a token to task *FD*. To avoid this situation, M_3 is partially switched to a parallel gateway, resulting in the creation of the exclusive merge gateway M_6, connected to task *LCDF* and M_3. This prevents a deadlock from occurring. Finally, M_2 is a valid candidate for being replaced by a parallel merge gateway, as the closest common ancestor of its parents is the parallel split gateway S_1. Thus, its type is switched to parallel. This process is then simulated in order to verify whether it can create deadlocks and/or livelocks. As it does not create any,

Algorithm 2 Algorithm for Generating the Most Parallel Process

Inputs: $G = (V, E, \Sigma)$ (BPMN Process), M_S (Split with Mandatory Merges)
Output: G_P (Most Parallel BPMN Process)

1: $S_{next} \leftarrow []$
2: $G_P \leftarrow \perp$
3:
4: **if** $\neg$hasDeadlockOrLivelock(G) **then**
5: $G_P \leftarrow$ getMostParallelProcessBetween(G, G_P)
6: **end if**
7:
8: **for** $v \in V$ **do**
9: **if** $\theta(v) =$  **then**
10: $G' \leftarrow$ copy(G) where $\theta(v) =$ ▷ v is a $\Rightarrow$ v becomes a
11: $S_{next} \leftarrow S_{next} \cup [G']$
12: **else if** $\theta(v) =$  **then**
13: **if** $M_S[v] \neq \perp$ **then** ▷ v is a mandatory merge $\Rightarrow$ v and $M_S[v]$ become 
14: $G' \leftarrow$ copy(G) where $\theta(v) =$ and $\theta(M_S[v]) =$
15: **else** ▷ v is not a mandatory merge $\Rightarrow$ only v becomes a
16: $G' \leftarrow$ copy(G) where $\theta(v) =$
17: **end if**
18: $S_{next} \leftarrow S_{next} \cup [G']$
19: **end if**
20: **end for**
21:
22: **for** $G' \in S_{next}$ **do**
23: $G_P \leftarrow$ getMostParallelProcessBetween(this(G'), G_P)
24: **end for**
25:
26: **return** G_P

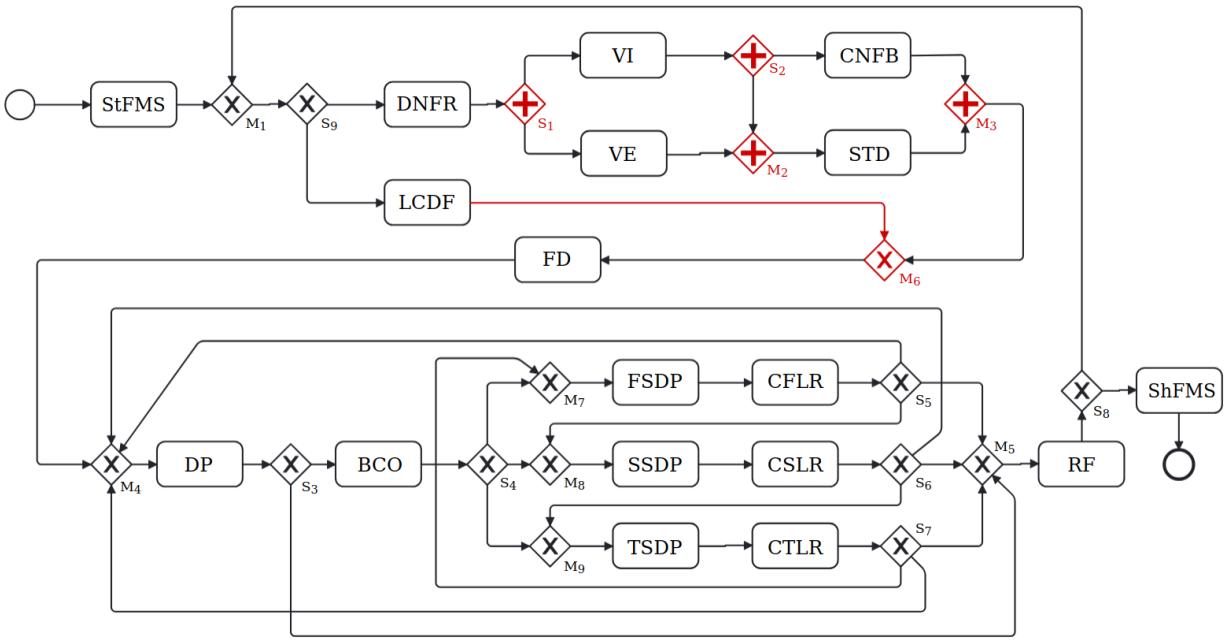


Figure 3.15: BPMN Process Resulting from the Parallelisation Phase

and is the most parallel BPMN process generated, it is kept as best candidate, and the generation of the BPMN process terminates.

3.9 Constraints Preservation

Beyond creating a BPMN process G compliant by construction with the constraints described in the original expressions, this approach also preserves the constraints satisfied by G at each step of its construction.

Theorem 3.1. *(Constraints Preservation) Let $G = (V, E, \Sigma)$ be the BPMN process built from the sequential constraints $Cons_1$ and enriched with exclusive gateways and start/end events, and let $Cons_2$, $Cons_3$, and $Cons_4$ be the sets of constraints respectively satisfied by G after steps 2 (Section 3.6), 3 (Section 3.7), and 4 (Section 3.8) of this approach. We state that $(\emptyset \subseteq) Cons_1 \subseteq Cons_2 \subseteq Cons_3 \subseteq Cons_4$.*

Proof (Sketch). This proof simply summarises all the verifications that were made during the construction of the BPMN process, which induce the validity of the aforementioned theorem.

- Step 1 creates G_1 out of all the sequential constraints present in the generated expressions. Trivially, $\emptyset \subseteq Cons_1$.
- Step 2 creates G_2 by enriching G_1 with the mutual exclusion constraints described in the expressions. As enriching G_1 with the mutual exclusion constraints only adds

new nodes/flows to G_1, $G_1 \subseteq G_2$. Thus, the sequential constraints satisfied by G_1 remain satisfied in G_2. Consequently, $Cons_1 \subseteq Cons_2$.

- Step 3 creates G_3 by enriching G_2 with the explicit loops described in the expressions. As enriching G_2 with the explicit loops only adds new nodes/flows to G_2, $G_2 \subseteq G_3$. Thus, the sequential constraints satisfied by G_2 remain satisfied in G_3. Moreover, a loop is added to G_2 if and only if it does not violate any mutual exclusion that it already satisfies. Consequently, $Cons_2 \subseteq Cons_3$.
- Step 4 creates G_4 by enriching G_3 with parallelism. As enriching G_3 with parallelism only adds new nodes/flows to G_3, or changes the type of some gateways, the sequential/loop constraints satisfied by G_3 remain satisfied in G_4. Moreover, a parallel split gateway is added to G_3 if and only if it does not violate any mutual exclusion satisfied by G_3. Consequently, $Cons_3 \subseteq Cons_4$.

□

3.10 Conclusion

In this chapter, we presented an approach aiming at generating business processes written in the BPMN format from textual descriptions of their behaviour. The approach takes as input the textual description, and, after several successive steps, returns a BPMN process that is syntactically correct with regards to the definition of the notation, and semantically correct with regards to the expressions returned by GPT. This correctness is ensured by several formal mechanisms, each of which is proven after its presentation. The approach was fully implemented and tested. More details about this are given in Section 5.1.

Chapter 4

Optimisation of BPMN Processes via Refactoring

"The biggest room in the world is the room for improvement."

Helmut Schmidt

Contents

Business process optimisation has become a strategic aspect of companies' management due to its potential of cost reduction and throughput improvement. To be able to optimise processes, there is a need to have at hand an explicit model of their behavior and quantitative features. Business processes are usually modelled using workflow-based notations, including an explicit description of execution time and resources associated with tasks. In this chapter, we propose process refactoring techniques whose goal is to change the structure of the process in order to optimise one or several criteria of interest such as process execution time, resource usage, or total costs. To do so, our approach consists of different ingredients including refactoring patterns, simulation techniques, and exploration of the near optimal process solutions.

This chapter presents three refactoring-based approaches, each of them having its strengths and its weaknesses, discussed at the end of their respective sections. Although differing on several aspects, these approaches share a common basis which is their internal representation. This representation allows us to define refactoring patterns, and to provide some strong semantics guarantees to them.

All these approaches were implemented in the form of 3 tools written in Java totaling 30k lines of code, for test and validation purposes. More details about their implementation and the corresponding experiments will be presented respectively in Sections 5.3.1, 5.3.2, and 5.3.3 of this manuscript.

Running example. The running example used throughout this section is the one depicted in Figure 4.1. It represents a simple goods delivery process, enriched with resources usage for the tasks and probabilities for conditional structures. For instance, task Deliver by drone requires one replica of resources driver and drone to execute, and has a probability of execution $p(\text{Deliver by drone}) = 0.4$, as it belongs to a branch of a choice structure having a probability of execution of 0.4. For now, the tasks do not have any duration, as they will vary depending on the approach being presented. Further information on these durations will thus be given in the next sections. Similarly, the IAT of the 100 instances of the process will depend on the approach. Lastly, the pool of available resources of the process, shared across all its running instances, and identical for all the presented approaches, is $P = \{\text{driver} \to 2, \text{drone} \to 2, \text{bike} \to 2, \text{employee} \to 6, \text{admin} \to 2, \text{daemon} \to 2\}$.

4.1 Sequence Graph

4.1.1 Definition

In this chapter, a BPMN process $G = (V, E, \Sigma)$ will not be represented as a classical graph, as in the previous chapter, but as a *sequence graph*. This notation has the main benefit of getting rid of the BPMN gateways by providing a hierarchical representation of the process, which facilitates the reasoning on the semantics preserved by the refactoring techniques.

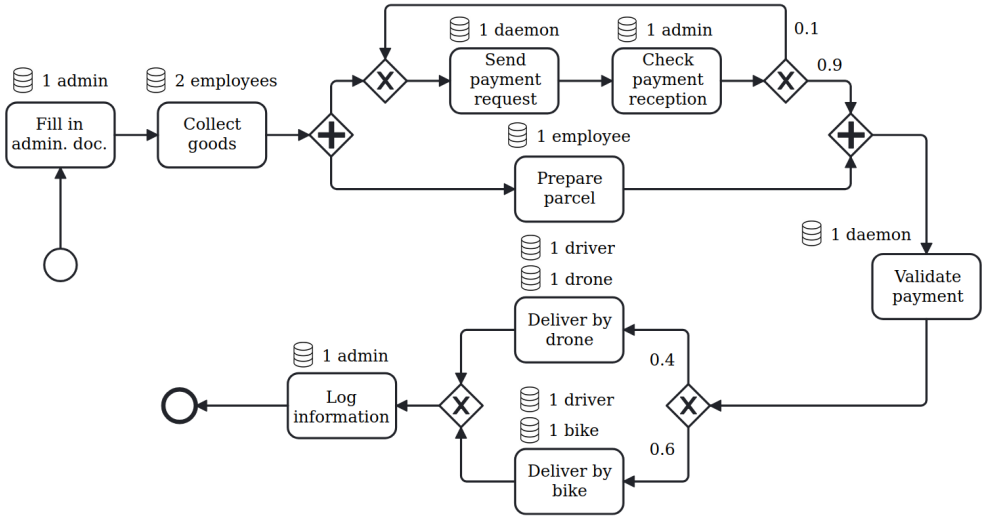


Figure 4.1: Running Example

Definition 4.1 (Sequence Graph)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process. The* sequence graph *representation of* G *is a hierarchical acyclic graph* $G_< = (V_<, E_<, \Sigma_<)$*, where:*

- $V_<$ *is a* set of vertices *whose elements* $v_<$ *are sets of* tasks, choice structures, loop structures, *and* sequence (sub)graphs *that correspond to parallel elements of* G*;*
- $E_<$ *is a* set of edges *connecting the vertices in* $V_<$ *such that each* $v_< \in V_<$ *has at most one incoming and one outgoing edge;*
- $\Sigma_< = \Sigma$.

Definition 4.2 (Choice Structure)**.** *Let* $G_< = (V_<, E_<, \Sigma_<)$ *be a sequence graph. For all* $v_< \in V_<$*, we define a* choice structure $\langle \times \rangle_C \in v_<$ *as a set of 2-tuples* $\{(G^1_<, p_1), ..., (G^n_<, p_n)\}$ *where, for all* $i \in [1...n]$*,* $G^i_<$ *is a* sequence graph*, and* $p_i \in [0, 1]$ *is its* probability *of execution.*

Definition 4.3 (Loop Structure)**.** *Let* $G_< = (V_<, E_<, \Sigma_<)$ *be a sequence graph. For all* $v_< \in V_<$*, we define a* loop structure $\langle \times \rangle_L \in v_<$ *as a 3-tuple* $(G^{FL}_<, G^{LF}_<, p_{LF})$ *where* $G^{FL}_<$ *and* $G^{LF}_<$ *are* sequence graphs*, and* $p_{LF} \in [0, 1]$ *is the* probability *of execution of* $G^{LF}_<$.

Remark 4.1. *The first subgraph of a loop structure, namely,* $G^{FL}_<$*, represents the body of the loop. As it is necessarily executed, its probability of execution is always 1, which is why it does not appear in the definition of this structure.*

To ease the usage of these sequence graphs, we introduce several *conventions* and useful *operators* that will be used throughout this chapter.

Definition 4.4 (Sequence Graph Conventions)**.** *Let* $G_< = (V_<, E_<, \Sigma_<)$ *be a sequence graph. The following* conventions *will be used in the rest of this thesis:*

- $\mathcal{L}$ *denotes the vertex type* loop*, corresponding to a loop structure of* $G_<$*;*

— $\mathcal{C}$ *denotes the vertex type* choice*, corresponding to a choice structure of* $G_<$*;*
— $\mathcal{T}$ *denotes the vertex type* task*;*
— $\mathcal{G}$ *denotes the vertex type* sequence graph*;*
— $\theta(v) \in \{\mathcal{L}, \mathcal{C}, \mathcal{T}, \mathcal{G}\}$ *returns the* type *of the vertex* v*, for all* $v_< \in V_<$*, for all* $v \in v_<$*.*

Definition 4.5 (Sequence Graphs Operators)**.** *Let* $G_< = (V_<, E_<, \Sigma_<)$ *be a sequence graph. We define the following* operators*:*

— $\texttt{first}(G_<) \stackrel{\text{def}}{=} v_< \in V_<$ *such that* $\nexists v'_< \in V_< \mid v'_< \to v_< \in E_<$*, returns the* first node *of* $G_<$*;*
— $\texttt{last}(G_<) \stackrel{\text{def}}{=} v_< \in V_<$ *such that* $\nexists v'_< \in V_< \mid v_< \to v'_< \in E_<$*, returns the* last node *of* $G_<$*;*
— $\texttt{pred}(v_<) \stackrel{\text{def}}{=} v'_< \in V_< \mid v'_< \to v_< \in E_<$*, returns the* predecessor node *of any* $v_< \in V_< \setminus \{\texttt{first}(G_<)\}$*;*
— $\texttt{succ}(v_<) \stackrel{\text{def}}{=} v'_< \in V_< \mid v_< \to v'_< \in E_<$*, returns the* successor node *of any* $v_< \in V_< \setminus \{\texttt{last}(G_<)\}$*;*
— $\mathcal{G}(G_<) \stackrel{\text{def}}{=} \bigcup_{v_< \in V_<} (S_{G_1} \cup S_{G_2} \cup S_{G_3})$ *returns all the* sequence graphs *hierarchically nested in* $V_<$*, with:*
  — $S_{G_1} = \{v \in v_< \mid \theta(v) = \mathcal{G}\}$ *(sequence graphs in the main nodes);*
  — $S_{G_2} = \bigcup_{\substack{v \in v_< \\ \theta(v) \in \{\mathcal{L},\mathcal{C}\}}} \bigcup_{v_i \in v} \{v_i\} \cup \mathcal{G}(v_i)$ *(sequence graphs composing loops/choices);*
  — $S_{G_3} = \bigcup_{\substack{v \in v_< \\ \theta(v) = \mathcal{G}}} \mathcal{G}(v)$ *(sequence graphs nested in sub-nodes).*
— $\mathcal{V}(G_<) \stackrel{\text{def}}{=} \bigcup_{G'_< = (V'_<, E'_<, \Sigma'_<) \in \mathcal{G}(G_<)} V'_<$*, returns* all the vertices *of the sequence graphs hierarchically nested in* $G_<$*;*
— $\mathcal{L}(G_<) \stackrel{\text{def}}{=} \bigcup_{v_< \in V_<} (S_{L_1} \cup S_{L_2} \cup S_{L_3})$ *returns all the* loop structures *hierarchically nested in* $V_<$*, with:*
  – $S_{L_1} = \{v \in v_< \mid \theta(v) = \mathcal{L}\}$ *(loop structures in the main nodes);*
  – $S_{L_2} = \bigcup_{\substack{v \in v_< \\ \theta(v) \in \{\mathcal{L},\mathcal{C}\}}} \bigcup_{v_i \in v} \mathcal{L}(v_i)$ *(loop structures inside loops/choices);*
  – $S_{L_3} = \bigcup_{\substack{v \in v_< \\ \theta(v) = \mathcal{G}}} \mathcal{L}(v)$ *(loop structures nested in sub-nodes).*
— $\mathcal{C}(G_<) \stackrel{\text{def}}{=} \bigcup_{v_< \in V_<} (S_{C_1} \cup S_{C_2} \cup S_{C_3})$ *returns all the* choice structures *hierarchically nested in* $V_<$*, with:*
  – $S_{C_1} = \{v \in v_< \mid \theta(v) = \mathcal{C}\}$ *(choice structures in the main nodes);*
  – $S_{C_2} = \bigcup_{\substack{v \in v_< \\ \theta(v) \in \{\mathcal{L},\mathcal{C}\}}} \bigcup_{v_i \in v} \mathcal{C}(v_i)$ *(choice structures inside loops/choices);*

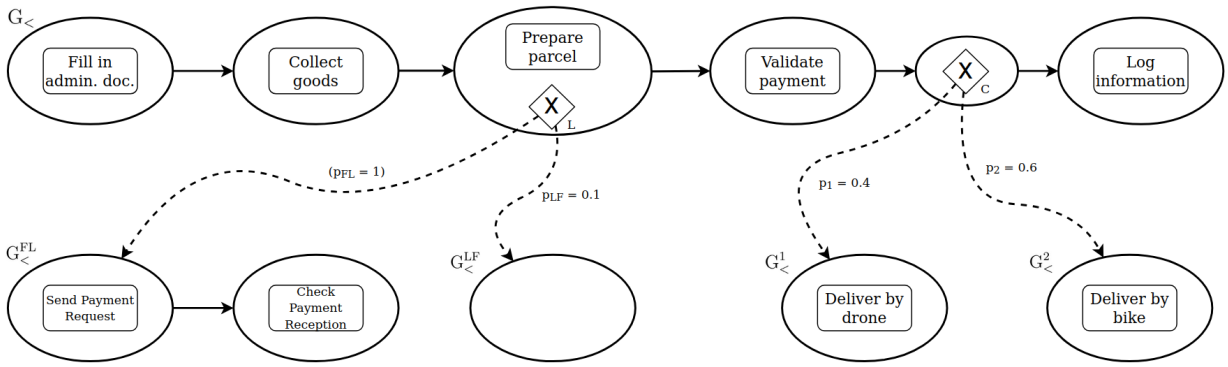


Figure 4.2: Sequence Graph Representation of the Running Example

- $S_{C_3} = \bigcup_{\substack{v \in v_< \\ \theta(v)=\mathcal{G}}} \mathcal{C}(v)$ *(choice structures nested in sub-nodes).*

— $\mathcal{S}(G_<) \stackrel{\text{def}}{=} \mathcal{L}(G_<) \cup \mathcal{C}(G_<)$ *returns all the* conditional structures *hierarchically nested in* $V_<$*;*

— $\mathcal{T}(G_<) \stackrel{\text{def}}{=} \bigcup_{G'_<=(V'_<,E'_<,\Sigma'_<)\in\mathcal{G}(G_<)} \bigcup_{v'_<\in V'_<} \{v \in v'_< \mid \theta(v) = \mathcal{T}\}$ *return* all the tasks *belonging to* $G_<$*;*

— $\Sigma(v_<) \stackrel{\text{def}}{=} \bigcup_{\substack{v\in v_< \\ \theta(v)=\mathcal{T}}} \{\sigma(v)\} \cup \bigcup_{\substack{v\in v_< \\ \theta(v)\in\{\mathcal{C},\mathcal{L}\}}} \bigcup_{G'_<=(V'_<,E'_<,\Sigma'_<)\in v} \Sigma'_< \cup \bigcup_{\substack{v\in v_< \\ \theta(v)=\mathcal{G}}} \Sigma_v$ *returns all the* labels *of the elements belonging to* $v_< \in V_<$*;*

— $\texttt{parent}(G'_<) \stackrel{\text{def}}{=} \begin{cases} v_< \in \mathcal{V}(G_<) \mid G'_< \in v_< & \text{if } G'_< \neq G_< \\ \bot & \text{otherwise} \end{cases} \quad \forall G'_< \in \mathcal{G}(G_<).$

Example. Figure 4.2 shows the sequence graph corresponding to the running example presented in Figure 4.1. As the reader can see, the main graph ($G_<$) consists of a sequence of six nodes. Four of them contain a single task. The third one contains a task and a loop structure $\langle \times \rangle_L$, itself consisting of two subgraphs: $G^{FL}_<$—with probability 1 and two nodes—corresponding to the body of the loop of the running example, and $G^{LF}_<$—with probability 0.1 and a single empty node—corresponding to the optional part of the loop. The fifth node contains a single choice structure $\langle \times \rangle_C$, itself consisting of two subgraphs: $G^1_<$ with probability 0.4 and $G^2_<$ with probability 0.6.

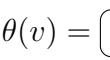
4.1.2 Sequence Graphs vs BPMN Processes

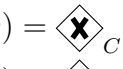
Generally speaking, sequence graphs cannot capture the expressiveness of BPMN processes, even restricted to the subset of the syntax handled in this thesis. This is intrinsically caused by the hierarchical structure of the sequence graphs, that prevents non-hierarchical BPMN structures from fitting into these bounds. However, a balanced BPMN process (see Figure 2.2(a)) can always be represented as a sequence graph. Indeed, each element of a

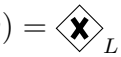
BPMN process has a *one-to-one mapping* with an element of a sequence graph.

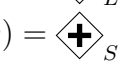
Proposition 4.1 (Sequence Graph and BPMN Processes Equivalence)**.** *Let* $G = (V, E, \Sigma)$ *be a balanced BPMN process. We state that there exists a sequence graph* $G_< = (V_<, E_<, \Sigma_<)$ *such that* G *and* $G_<$ *are* semantically equivalent*, written* $G \sim G_<$*.*

Proof (Sketch). Let $G = (V, E, \Sigma)$ be a balanced BPMN process. The idea of this proof is to show that every element of G can be (uniquely) mapped to an element of a well-suited sequence graph $G_< = (V_<, E_<, \Sigma_<)$, and vice-versa. This mapping, represented by the $\rightsquigarrow$ symbol, is equivalent to a bijection between the set of balanced BPMN processes and the set of sequence graphs, thus proving the desired equivalence. Let us present the main lines of the proof:

— $\forall v \in V \mid \theta(v) =$  , $\exists t_< \in \mathcal{T}(G_<) \mid v \rightsquigarrow t_<$;

— $\forall v \in V \mid \theta(v) =$  , $\exists c_< \in \mathcal{C}(G_<) \mid v \rightsquigarrow c_<$;

— $\forall v \in V \mid \theta(v) =$  , $\exists l_< \in \mathcal{L}(G_<) \mid v \rightsquigarrow l_<$;

— $\forall v \in V \mid \theta(v) =$  , $\exists v_< \in \mathcal{V}(G_<) \mid v \rightsquigarrow v_<$.

The same idea holds for the reverse way: each element of a sequence graph can be written in BPMN. Thus, balanced BPMN processes have an equivalent sequence graph representation, and vice-versa. □

4.2 Notions & Operations on Sequence Graphs

The main idea of the refactoring approach is to modify syntactically (and thus, semantically) the sequence graph of a process, in order to optimise it. Thus, this subsection presents several notions and operations on sequence graphs that will be used in the rest of this chapter.

The first notion introduced in this section is the notion of *closest sequence graph* of a task. The closest sequence graph of a task is the only sequence graph containing a node itself containing the given task. Such a graph is unique under the assumption that each task of the process occurs once and only once. However, this restriction is rather light as, for instance, indexing the tasks of the process before applying refactoring operations would be sufficient.

Definition 4.6 (Closest Sequence Graph)**.** *Let* $G_< = (V_<, E_<, \Sigma_<)$ *be a sequence graph. For all* $t \in \mathcal{T}(G_<)$*, there exists a unique* $G'_< = (V'_<, E'_<, \Sigma'_<) \in \mathcal{G}(G_<)$ *such that there is a* $v'_< \in V'_<$ *for which* $t \in v'_<$*. This sequence graph* $G'_<$ *is the* closest sequence graph *of* t*, and can be retrieved with the operator* $\mathcal{G}^*(t)$*.*

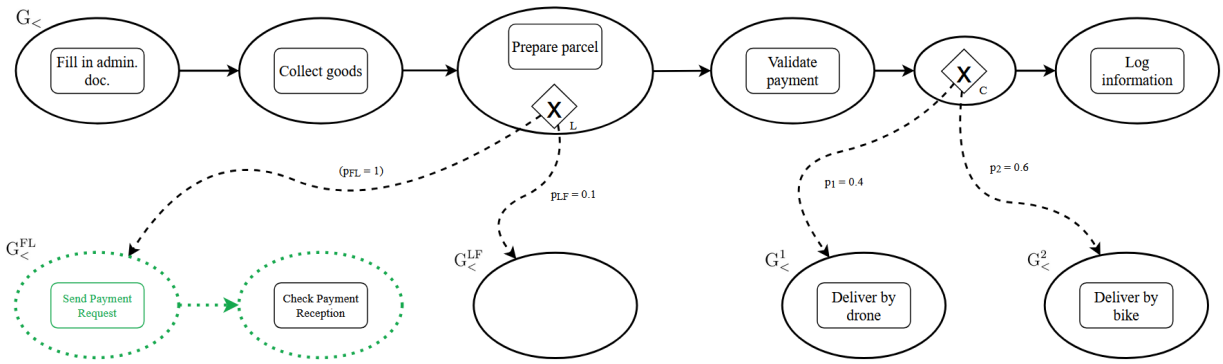


Figure 4.3: Illustration of the Closest Sequence Graph of Task Send Payment Request

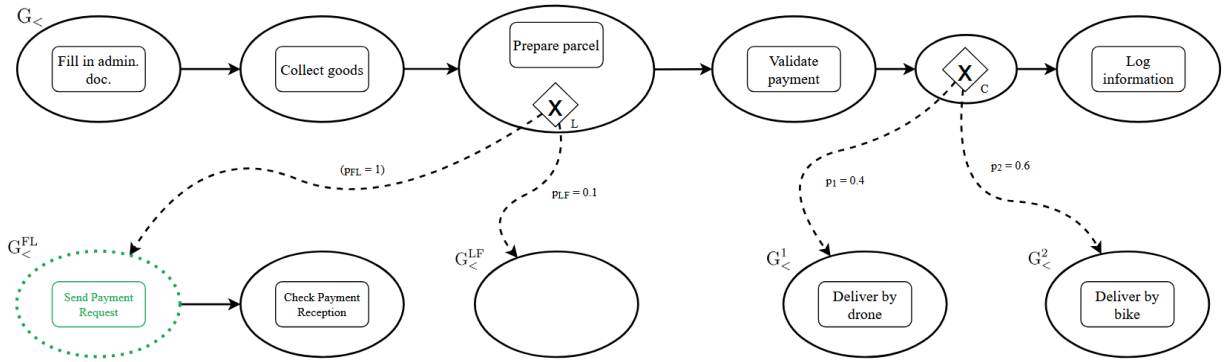


Figure 4.4: Illustration of the Closest Node of Task Send Payment Request

Example. Figure 4.3 shows the closest sequence graph of task Send Payment Request. This graph is the subgraph $G^{FL}_{<}$ belonging to the loop structure of the process, presented in green dotted lines, as it contains the desired task in one of its nodes.

The node of the closest sequence graph of a task containing that task is itself called the *closest node* of the task.

Definition 4.7 (Closest Node)**.** *Let* $G_< = (V_<, E_<, \Sigma_<)$ *be a sequence graph. For all* $t \in \mathcal{T}(G_<)$*, the* closest node *of* t *is defined as*

$$\mathcal{V}^*(t) \stackrel{\text{def}}{=} v_< \in \mathcal{G}^*(t) \mid t \in v_<$$

Example. Figure 4.4 shows the closest node of task Send Payment Request. This node is the first node of the subgraph $G^{FL}_{<}$ belonging to the loop structure of the process, presented in green dotted lines, as it contains the desired task in its set of elements.

Similarly, the *closest conditional structure* of a task is a conditional structure containing that task, and such that there is no nested conditional structure of this structure containing that task. If the task does not belong to any such structure, its closest conditional structure

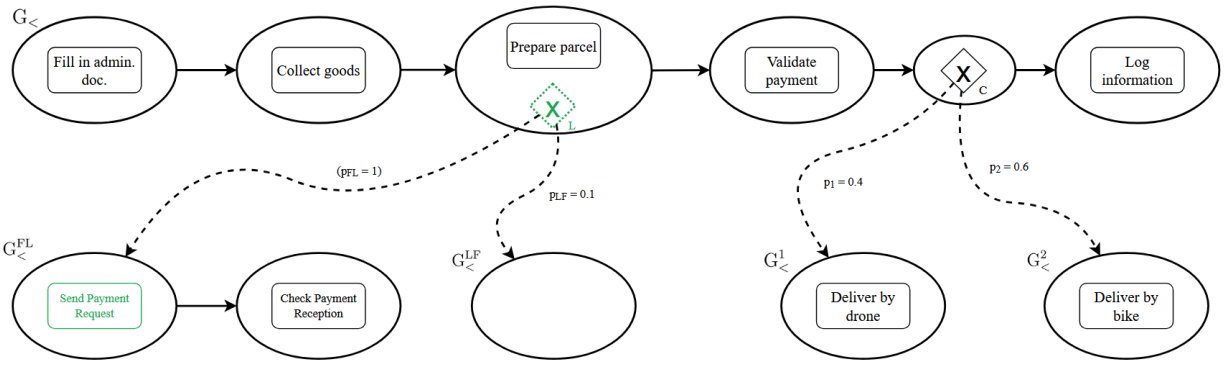


Figure 4.5: Illustration of the Closest Conditional Structure of Task Send Payment Request

is undefined.

Definition 4.8 (Closest Conditional Structure of a Task)**.** *Let* $G_< = (V_<, E_<, \Sigma_<)$ *be a sequence graph. For all* $t \in \mathcal{T}(G_<)$*, we define the* closest conditional structure *of* t *as:*

$$\mathcal{S}^*(t) \stackrel{\text{def}}{=} \begin{cases} s & \text{if } (\exists s \in \mathcal{S}(G_<),\ \exists G^i_< \in s,\ \exists G'_< \in \mathcal{G}(G^i_<),\ \exists v'_< \in G'_< \text{ such that } t \in v'_<) \\ & \wedge\ (\forall s' \in \mathcal{S}(G'_<),\ \forall G^{i'}_< \in s',\ \forall G''_< \in \mathcal{G}(G^{i'}_<),\ \nexists v''_< \in G''_< \text{ such that } t \in v''_<) \\ \bot & \text{otherwise} \end{cases}$$

Example. Figure 4.5 shows the closest conditional structure of task Send Payment Request. As this task belongs to the loop structure $\overset{\text{X}}{\diamond}_L$, it admits as closest conditional structure this loop, displayed in green dotted lines in the figure.

In order to preserve the semantics of the original BPMN process, a task that is subject to move must not cross its *boundary*, that is, the sequence graph of the closest conditional structure containing that task, or the entire sequence graph if the task does not admit any closest conditional structure.

Definition 4.9 (Task Boundary)**.** *Let* $G_< = (V_<, E_<, \Sigma_<)$ *be a sequence graph. For all* t *in* $\mathcal{T}(G_<)$*, we define the* boundary *of* t *as:*

$$\partial(t) \stackrel{\text{def}}{=} \begin{cases} G^i_< \in \mathcal{S}^*(t) \mid t \in \mathcal{T}(G^i_<) & \text{if } \mathcal{S}^*(t) \neq \bot \\ G_< & \text{otherwise} \end{cases}$$

Example. In the particular case of our running example (Figure 4.2), the boundary of a task strictly matches its closest sequence graph. Thus, the boundary of task Send Payment Request corresponds to the sequence graph highlighted in green dotted lines in Figure 4.3.

Remark 4.2. *Let* $G_< = (V_<, E_<, \Sigma_<)$ *be a sequence graph. The sequence graph containing the closest node of any task of* $G_<$ *is a subgraph of the boundary of that task. More formally,*

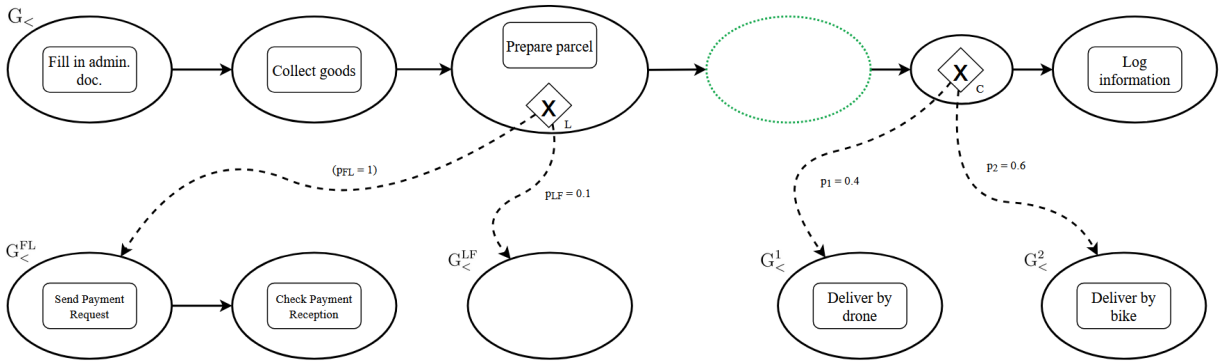


Figure 4.6: Illustration of the Removal of Task Send Payment Request

for all $t \in \mathcal{T}(G_<),\ G'_< = (V'_<, E'_<, \Sigma'_<) \in \mathcal{G}(G_<)$ *such that* $\mathcal{V}^*(t) \in V'_<$ *is a subgraph of* $\partial(t)$.

The movement of a task under its boundary is actually performed by two successive steps: the *removal* of that task, and its *(re)insertion*.

Definition 4.10 (Task Removal)**.** *Let* $G_< = (V_<, E_<, \Sigma_<)$ *be a sequence graph. For all* t *in* $\mathcal{T}(G_<)$*, we define the* removal of task t *as the alteration of* $G_<$ *such that* $\mathcal{V}^*(t) = \mathcal{V}^*(t) \setminus \{t\}$*. This ensured by the operator* $\texttt{rem}(G_<, t) = \widehat{G}_<$.

Example. The removal of task Send Payment Request lets the sequence graph of our running example with an empty node, as depicted in green dotted lines in Figure 4.6.

After a task removal, the sequence graph may contain some useless constructs. For instance, a task removal may lead to a node containing a single sequence graph, which can thus be extracted from this node, or to an empty node, or to a sequence graph containing a single non-empty node, which consequently becomes useless and can be removed. To remove these useless constraints from the graph, it is simplified after every task removal. These simplifications of the sequence graph, repeated as many times as necessary to obtain a sequence graph in which no simplification can be performed, are unified under the term *normalisation*.

Definition 4.11 (Sequence Graph Normalisation)**.** *Let* $G_< = (V_<, E_<, \Sigma_<)$ *be a sequence graph. The* normalisation *operation, written* $\texttt{norm}(G_<)$*, is applied repeatedly on* $G_<$ *after any task removal until reaching its idempotence, that is,* $\texttt{norm}(G_<) = G_<$*. This operation is defined as*

$$\texttt{norm}(G_<) \stackrel{\text{def}}{=} \bigcirc_{i \in [1\ldots3]} \texttt{norm}_i(G_<)$$

where

— $\texttt{norm}_1(G_<) \stackrel{\text{def}}{=} \begin{cases} \widehat{G_<} & \text{if } \exists G'_< = (V'_<, E'_<, \Sigma'_<) \in \mathcal{G}(G_<), \exists v'_< \in V'_< \mid v'_< = \emptyset \\ G_< & \text{otherwise} \end{cases}$

where $\widehat{G_<}$ *is the alteration of* $G_<$ *in which* $G'_< = (\widehat{V'_<}, \widehat{E'_<}, \Sigma')$ *such that:*

– $\widehat{V'_<} = V'_< \setminus \{v'_<\}$;

– $\widehat{E'_<} = \begin{cases} \bullet\ E'_< \setminus \{\texttt{pred}(v'_<) \to v'_<, v'_< \to \texttt{succ}(v'_<)\} \cup \{\texttt{pred}(v'_<) \to \texttt{succ}(v'_<)\} \\ \quad \text{if } v'_< \neq \texttt{last}(G'_<) \ \wedge\ v'_< \neq \texttt{first}(G'_<); \\ \bullet\ E'_< \setminus \{\texttt{pred}(v'_<) \to v'_<\} \quad \text{if } v'_< \neq \texttt{first}(G'_<); \\ \bullet\ E'_< \setminus \{v_< \to \texttt{succ}(v'_<)\} \quad \text{if } v'_< \neq \texttt{last}(G'_<); \\ \bullet\ E'_< \quad \text{otherwise.} \end{cases}$

(removal of the empty nodes)

— $\texttt{norm}_2(G_<) \stackrel{\text{def}}{=} \begin{cases} \widehat{G_<} & \text{if } \exists G'_< \in \mathcal{G}(G_<), \exists v'_< \in V'_< \mid |v'_<| = 1 \ \wedge\ \theta(\texttt{any}(v'_<)) = \mathcal{G} \\ G_< & \text{otherwise} \end{cases}$

where $G'_< = (V'_<, E'_<, \Sigma'_<)$, $\theta(\texttt{any}(v'_<))$ *can be written* $G''_< = (V''_<, E''_<, \Sigma''_<)$ *and* $\widehat{G_<}$ *is the alteration of* $G_<$ *in which* $G'_< = (\widehat{V'_<}, \widehat{E'_<}, \Sigma')$ *such that:*

– $\widehat{V'_<} = V'_< \setminus \{v'_<\} \cup V''_<$;

– $\widehat{E'_<} = \begin{cases} \bullet\ E'_< \setminus \{\texttt{pred}(v'_<) \to v'_<, v'_< \to \texttt{succ}(v'_<)\} \cup \{\texttt{pred}(v'_<) \to \texttt{first}(G''_<), \\ \quad \texttt{last}(G''_<) \to \texttt{succ}(v'_<)\} \cup E''_< \ \text{ if } v'_< \neq \texttt{last}(G'_<) \ \wedge\ v'_< \neq \texttt{first}(G'_<); \\ \bullet\ E'_< \setminus \{\texttt{pred}(v'_<) \to v'_<\} \cup \{\texttt{pred}(v'_<) \to \texttt{first}(G''_<)\} \cup E''_< \\ \quad \text{if } v'_< \neq \texttt{first}(G'_<); \\ \bullet\ E'_< \setminus \{v_< \to \texttt{succ}(v'_<)\} \cup \{\texttt{last}(G''_<) \to \texttt{succ}(v'_<)\} \cup E''_< \\ \quad \text{if } v'_< \neq \texttt{last}(G'_<); \\ \bullet\ E'_< \cup E''_< \quad \text{otherwise.} \end{cases}$

(removal of the useless nodes)

— $\texttt{norm}_3(G_<) \stackrel{\text{def}}{=} \begin{cases} \widehat{G_<} \text{ if } \exists G'_< \in \mathcal{G}(G_<) \ s.t.\ |V'_<| \leq 1 \wedge G'_< \neq G_< \wedge \forall s \in \mathcal{S}(G_<), G'_< \notin s \\ G_< \text{ otherwise} \end{cases}$

where $G'_< = (V'_<, E'_<, \Sigma'_<)$, *and* $\widehat{G_<}$ *is the alteration of* $G_<$ *such that* $\texttt{parent}(G'_<) = \texttt{parent}(G'_<) \setminus \{G'_<\} \cup V'_<$ *(removal of the useless sequence graphs).*

Example. Given the sequence graph obtained after removing task Send Payment Request from the original sequence graph, the normalisation rule $\texttt{norm}_1$ will remove the empty node of this graph (shown in green dotted lines in Figure 4.6) and produce the sequence graph presented in Figure 4.7.

A task that has been removed from a sequence graph will at some point be inserted back

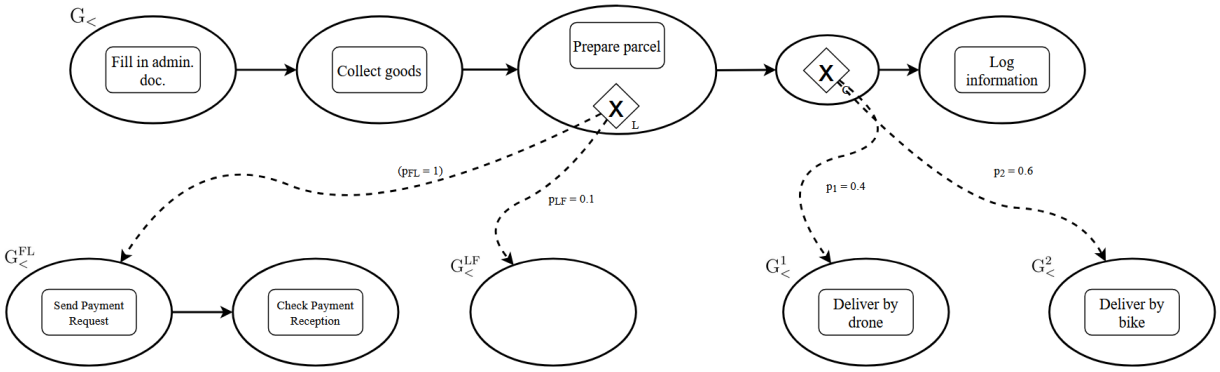


Figure 4.7: Illustration of the Normalisation Rule $\texttt{norm}_1$ Applied to the Sequence Graph of Figure 4.6

into it. This (re)insertion, that must occur under the boundary of the given task to preserve the original trace semantics of the process, is ensured by four patterns whose goal is to place this task at every possible position of the sequence graph. Each of these four patterns generates a new sequence graph in which the position of the task differs from the others.

Definition 4.12 (Task (Re)Insertion)**.** *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, let $t \notin \mathcal{T}(G_<)$ be a task already removed from $G_<$, and let $G'_< = \partial(t)$. We define the set of sequence graphs generated by* (re)inserting *t in $G_<$ as:*

$$\texttt{ins}(G_<, t) = \bigcup_{\widehat{G}'_< \in \texttt{gen}(G'_<, t)} \{\widehat{G}_<\}$$

where $\widehat{G}_<$ is the alteration of $G_<$ such that $\texttt{parent}(G'_<) = \texttt{parent}(G'_<) \setminus \{G'_<\} \cup \{\widehat{G}'_<\}$.

Remark 4.3. *In the case where the main sequence graph $G_<$ is the boundary of the task t, namely, $\partial(t)$, the (re)insertion operation replaces the main sequence graph itself, as it does not have any parent.*

Definition 4.13 (Generation of Sequence Graphs)**.** *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, let $t \notin \mathcal{T}(G_<)$ be a task already removed from $G_<$, and let $G'_< = (V'_<, E'_<, \Sigma'_<) = \partial(t)$. We define the modified versions of $G'_<$ obtained by (re)inserting t in it as:*

$$\texttt{gen}(G'_<, t) \stackrel{\text{def}}{=} \bigcup_{i \in [1...4]} \texttt{gen}_{P_i}(G'_<, t) \;\cup \bigcup_{v'_< \in V'_<} \bigcup_{\substack{v \in v'_< \\ \theta(v) = \mathcal{G}}} \texttt{gen}(v, t)$$

Definition 4.14 (Pattern 1)**.** *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, let $t \notin \mathcal{T}(G_<)$ be a task already removed from $G_<$, let $G'_< = (V'_<, E'_<, \Sigma'_<) = \partial(t)$, and let $\tilde{v} = \{t\}$ be a new sequence node containing only t. The set of sequence graphs generated by applying*

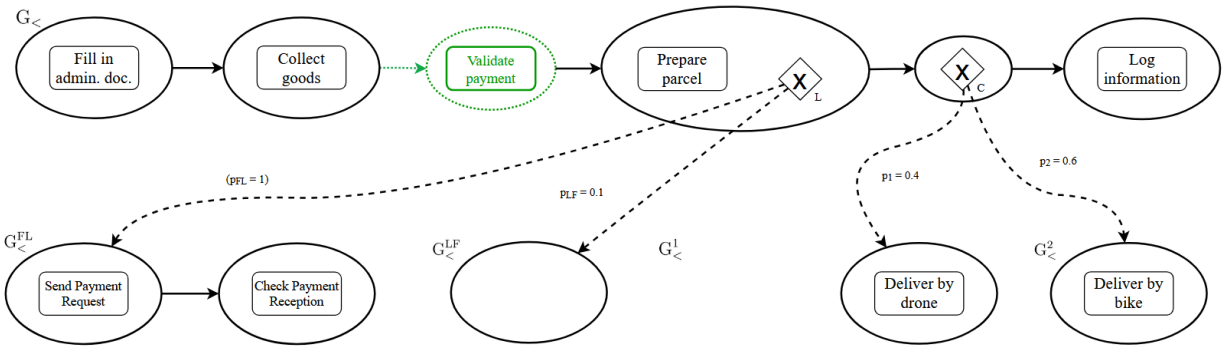


Figure 4.8: Illustration of Pattern 1 on Task Validate Payment

Pattern 1 *to* $G'_<$ *and* t *is defined as:*

$$\mathtt{gen}_{P_1}(G'_<, t) \stackrel{\text{def}}{=} \begin{cases} (\{\tilde{v}\}, \emptyset) & \text{if } V'_< = \emptyset \\ \bigcup\limits_{i \in [1...5]} \mathtt{gen}^i_{P_1}(G'_<, t) & \text{otherwise} \end{cases}$$

where:

— $\mathtt{gen}^1_{P_1}(G'_<, t) \stackrel{\text{def}}{=} \bigcup\limits_{v_j \to v_k \in E'_<} (V'_< \cup \{\tilde{v}\}, E'_< \setminus \{v_j \to v_k\} \cup \{v_j \to \tilde{v}, \tilde{v} \to v_k\}, \Sigma'_< \cup \{\sigma(t)\})$ *represents the addition of* $\tilde{v}$ *in sequence with any other node of* $G'_<$*;*

— $\mathtt{gen}^2_{P_1}(G'_<, t) \stackrel{\text{def}}{=} \{(V'_< \cup \{\tilde{v}\}, E'_< \cup \{\tilde{v} \to \mathtt{first}(G'_<)\}, \Sigma'_< \cup \{\sigma(t)\})\}$ *represents the addition of* $\tilde{v}$ *in sequence before the first node of* $G'_<$*;*

— $\mathtt{gen}^3_{P_1}(G'_<, t) \stackrel{\text{def}}{=} \{(V'_< \cup \{\tilde{v}\}, E'_< \cup \{\mathtt{last}(G'_<) \to \tilde{v}\}, \Sigma'_< \cup \{\sigma(t)\})\}$ *represents the addition of* $\tilde{v}$ *in sequence after the last node of* $G'_<$*;*

— $\mathtt{gen}^4_{P_1}(G'_<, t) \stackrel{\text{def}}{=} \bigcup\limits_{v'_< \in V'_<} \bigcup\limits_{\substack{v' \in v'_< \\ \theta(v') = \mathtt{task}}} (V'_< \setminus v'_< \cup \{v'_< \setminus \{v'\}\} \cup \{(\{\{v'\}, \tilde{v}\}, \{\tilde{v} \to \{v'\}\}, \{\sigma(v'), \sigma(t)\})\}, E'_<, \Sigma'_< \cup \{\sigma(t)\})$ *represents the addition of* $\tilde{v}$ *in sequence before any task of any node of* $G'_<$*;*

— $\mathtt{gen}^5_{P_1}(G'_<, t) \stackrel{\text{def}}{=} \bigcup\limits_{v'_< \in V'_<} \bigcup\limits_{\substack{v' \in v'_< \\ \theta(v') = \mathtt{task}}} (V'_< \setminus v'_< \cup \{v'_< \setminus \{v'\}\} \cup \{(\{\{v'\}, \tilde{v}\}, \{\{v'\} \to \tilde{v}\}, \{\sigma(v'), \sigma(t)\})\}, E'_<, \Sigma'_< \cup \{\sigma(t)\})$ *represents the addition of* $\tilde{v}$ *in sequence after any task of any node of* $G'_<$*;*

Example. Figure 4.8 illustrates a possible result of applying Pattern 1 to task Validate payment. As the reader can see, the task Validate payment has been put inside a new sequence node (in green dotted line), which was inserted between other nodes of the graph.

Pattern 2 consists in putting the task to insert in parallel with any *non-empty subsequence of nodes* of the graph.

Definition 4.15 (Non-Empty Subsequence of Nodes)**.** *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph. A non-empty subsequence of nodes of $G_<$ is any $\{v^1_<, ..., v^n_<\} \subseteq V_<$ such that:*

— $\{v^1_<, ..., v^n_<\} \neq \emptyset$;
— $\forall i \in [1...n-1],\ v^i_< \rightarrow v^{i+1}_< \in E_<$.

The set of all non-empty subsequences of nodes of $G_<$ can be retrieved with the operator $\mathcal{P}^*_O(G_<)$.

Definition 4.16 (Pattern 2)**.** *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, let $t \notin \mathcal{T}(G_<)$ be a task already removed from $G_<$, and let $G'_< = \partial(t)$. The set of sequence graphs generated by applying* Pattern 2 *to $G'_<$ and t is defined as*

$$\texttt{gen}_{P_2}(G'_<, t) \stackrel{\text{def}}{=} \bigcup_{\substack{s \in \mathcal{P}^*_O(G'_<) \\ s=(v^1_<,...,v^n_<)}} \begin{cases} \bullet\ (\{\tilde{v}\}, \emptyset, \Sigma'_<) \quad \textbf{if} \quad v^1_< = \texttt{first}(G'_<) \ \wedge\ v^n_< = \texttt{last}(G'_<) \\ \bullet\ (V'_< \setminus s \ \cup\ \tilde{v}, E'_< \setminus \bigcup_{i \in [1...n-1]} \{v^i_< \rightarrow v^{i+1}_<\} \setminus \{v^n_< \rightarrow \texttt{succ}(v^n_<)\} \\ \quad \cup \{\tilde{v} \rightarrow \texttt{succ}(v^n_<)\}, \Sigma'_<) \quad \textbf{if} \quad v^1_< = \texttt{first}(G'_<) \\ \bullet\ (V'_< \setminus s \ \cup\ \tilde{v}, E'_< \setminus \bigcup_{i \in [1...n-1]} \{v^i_< \rightarrow v^{i+1}_<\} \setminus \{\texttt{pred}(v^n_<) \rightarrow v^n_<\} \\ \quad \cup \{\texttt{pred}(v^n_<) \rightarrow \tilde{v}\}, \Sigma'_<) \quad \textbf{if} \quad v^n_< = \texttt{last}(G'_<) \\ \bullet\ (V'_< \setminus s \ \cup\ \tilde{v}, E'_< \setminus \bigcup_{i \in [1...n-1]} \{v^i_< \rightarrow v^{i+1}_<\} \\ \quad \setminus \{\texttt{pred}(v^n_<) \rightarrow v^n_<, v^n_< \rightarrow \texttt{succ}(v^n_<)\} \\ \quad \cup \{\texttt{pred}(v^n_<) \rightarrow \tilde{v}, \tilde{v} \rightarrow \texttt{succ}(v^n_<)\}, \Sigma'_<) \quad \textbf{otherwise} \end{cases}$$

where $\tilde{v} = \{\{t, (s, \bigcup_{i \in [1...n-1]} \{v^i_< \rightarrow v^{i+1}_<\}, \bigcup_{i \in [1...n]} \Sigma(v^i_<))\}\}$.

Example. Figure 4.9 illustrates a possible result of applying Pattern 2 to task Validate payment. As the reader can see, the task Validate payment has been put inside a new sequence node (in green dotted line), in parallel of a non-empty subsequence of nodes of the graph that now belongs to this new node.

Pattern 3 consists in putting the task to insert before or after any combination of elements of any node of the graph.

Definition 4.17 (Pattern 3)**.** *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, let $t \notin \mathcal{T}(G_<)$ be a task already removed from $G_<$, and let $G'_< = \partial(t)$. The set of sequence graphs generated by applying* Pattern 3 *to $G'_<$ and t is defined as:*

$$\texttt{gen}_{P_3}(G'_<, t) \stackrel{\text{def}}{=} \bigcup_{i \in [1...2]} \texttt{gen}^i_{P_3}(G'_<, t)$$

where:

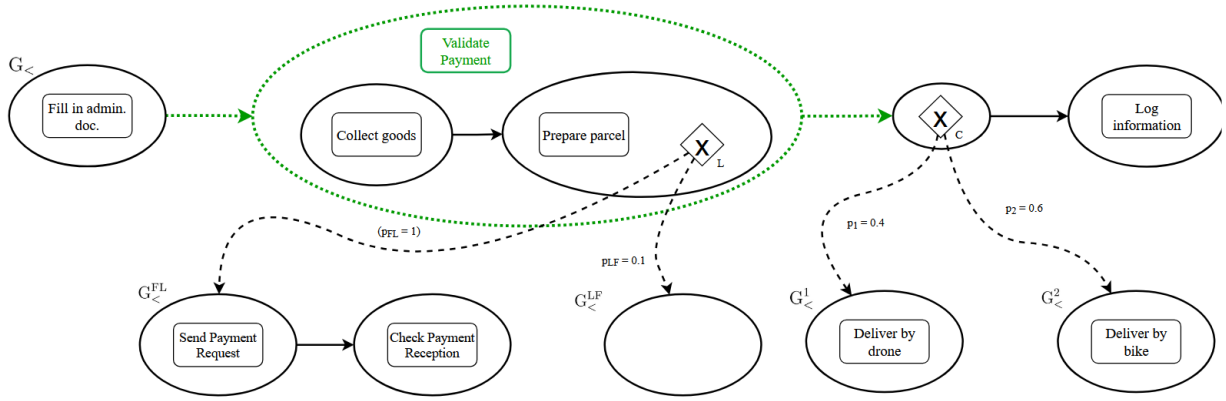


Figure 4.9: Illustration of Pattern 2 on Task Validate Payment

— $\mathsf{gen}^1_{P_3}(G'_<, t) \stackrel{\text{def}}{=} \bigcup_{v'_< \in V'_<} \bigcup_{\substack{S_{v'_<} \in 2^{v'_<} \\ |S_{v'_<}| < |v'_<|}} (V'_< \setminus v'_< \cup \{v'_< \setminus S_{v'_<}, (\{\{t\}, S_{v'_<}\}, \{\{t\} \to S_{v'_<}\}, \{\sigma(t) \cup \bigcup_{v''_< \in S_{v'_<}} \Sigma(v''_<)\})\})$ *represents the addition of task* t *before any combination of elements of any node of* $G'_<$*;*

— $\mathsf{gen}^2_{P_3}(G'_<, t) \stackrel{\text{def}}{=} \bigcup_{v'_< \in V'_<} \bigcup_{\substack{S_{v'_<} \in 2^{v'_<} \\ |S_{v'_<}| < |v'_<|}} (V'_< \setminus v'_< \cup \{v'_< \setminus S_{v'_<}, (\{\{t\}, S_{v'_<}\}, \{S_{v'_<} \to \{t\}\}, \{\sigma(t) \cup \bigcup_{v''_< \in S_{v'_<}} \Sigma(v''_<)\})\})$ *represents the addition of task* t *after any combination of elements of any node of* $G'_<$*.*

Example. Figure 4.10 shows a possible result of applying Pattern 3 to task Validate payment while releasing the constraint on the minimal size of the combination. As the reader can see, the task Validate payment has been put inside a new sequence node (in green dotted line), connected to a node containing the task Prepare parcel (which is also new). Here, the task Prepare parcel acts as a combination of size 1 of elements of the node containing it and the loop construct.

Pattern 4 deals with choice structures, and, consequently, slightly differs from the three previous ones. A choice is a structure composed of several branches, among which only one will be executed. To ensure that the traces of the process remain unchanged and that the frequency of task occurrences within them is preserved, choice structures require a specific treatment. The only way to properly insert a task inside a choice is to insert it in all the branches of the choice. By doing so, the execution of such specific task remains unconditional. To perform this insertion, the four patterns are applied to each sequence graph composing the choice, thus creating several new sequence graphs for each branch of the choice. The cartesian product of these sets of sequence graphs is then computed to obtain a set of unique choice structures, each corresponding to a different set of previously

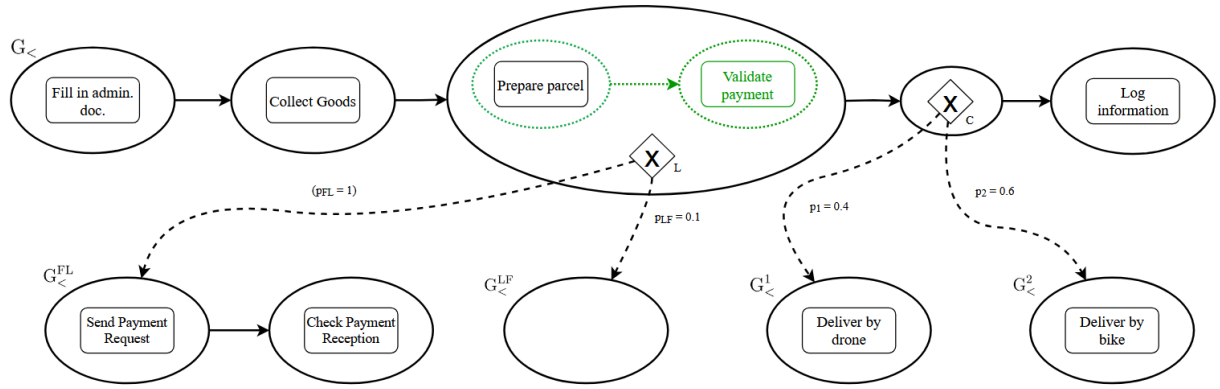


Figure 4.10: Illustration of Pattern 3 on Task Validate Payment

generated sequence graphs. Each of these choice structures then replaces the original one to create a new complete sequence graph.

Definition 4.18 (Pattern 4)**.** *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, let $t \notin \mathcal{T}(G_<)$ be a task already removed from $G_<$, and let $G'_< = \partial(t)$. The set of sequence graphs generated by applying* Pattern 4 *to $G'_<$ and t is defined as:*

$$\texttt{gen}_{P_4}(G'_<, t) \stackrel{\text{def}}{=} \bigcup_{v'_< \in V'_<} \bigcup_{\substack{v \in v'_< \\ \theta(v)=\mathcal{C} \\ v=((G^1_<,p_1),\ldots,(G^n_<,p_n))}} \bigcup_{(\widehat{G^1_<},\ldots,\widehat{G^n_<}) \in \prod_{i=1}^{n} \texttt{gen}(G^i_<,t)} \{(\widehat{V'_<}, E'_<, \Sigma'_< \cup \{\sigma(t)\})\}$$

where $\widehat{V'_<} = V'_< \setminus \{v'_<\} \cup \{v'_< \setminus \{v\} \cup \{((\widehat{G^1_<}, p_1), \ldots, (\widehat{G^n_<}, p_n))\}\}$.

Example. Figure 4.11 shows a possible result of applying Pattern 4 to task Validate payment. As the reader can see, the task Validate payment has entered the choice structure $\times_C$ and is now appearing in both subgraphs (branches) of that choice. In the first subgraph ($G^1_<$), the task has been inserted after the task Validate payment by Pattern 1, whereas in the second subgraph ($G^2_<$), it has been put in parallel of task Deliver by bike by Pattern 2.

4.3 Structure and Trace Persistency

This section demonstrates that moving a task within a sequence graph (i.e., removing it and inserting it back later) preserves its structure and its traces, excluding the moved task. More precisely, we show that for every trace in the original graph, there exists a corresponding trace in the modified graph that is a permutation of the original one, maintaining the same task occurrences. However, before presenting these results, we need some considerations and some preliminary results.

For some of the results in this section, we want to remove all the replicas of a given task introduced by the `gen` operation, and more precisely, by Pattern 4. Indeed, as a

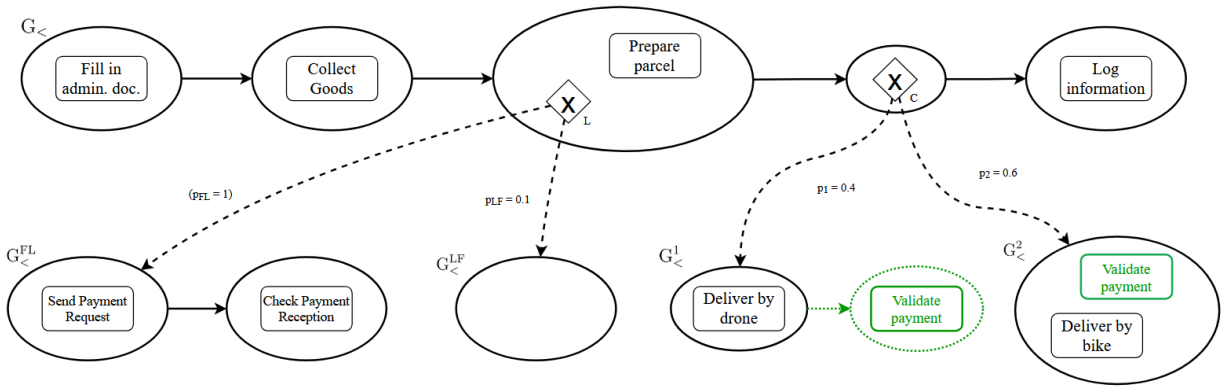


Figure 4.11: Illustration of Pattern 4 on Task Validate Payment

consequence of potentially inserting the task to move inside a choice structure, this pattern may generate several replicas of this given task, in order to insert each of them in a different branch of the choice structure. We thus introduce a slightly modified version of the `remove` operator, that is in charge of removing all the replicas of a given task, based on its name.

Definition 4.19 (Complete Removal of a Task)**.** *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, and let $t \in \mathcal{T}(G_<)$ be a task of $G_<$. We define the* $\texttt{rem}^*$ *operator, which removes all occurrences of t in G, as the composition of the* `rem` *operation on all the tasks having the same label than t, that is*

$$\texttt{rem}^*(G_<, t) \stackrel{\text{def}}{=} \bigcirc_{\substack{t' \in \mathcal{T}(G_<) \\ \sigma(t') = \sigma(t)}} \texttt{rem}(G_<, t')$$

Additionally, the comparison of the structure of the sequence graphs without the moved task is simpler if these sequence graphs are normalised. Thus, we consider that the normalisation operations defined in Section 4.2 are applied right after any task removal.

Our first result states that, up to the normalisation operations, apart from moving the task from one place to another, the structure of the graph is not modified by the `ins` operation. Before proving it, let us show that the statement is true under the boundary of the task being moved, preparing the field for considering the whole sequence graph instead of only the boundary.

Proposition 4.2 (Boundary Persistency)**.** *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, let $t \notin \mathcal{T}(G_<)$ be a task already removed from $G_<$, and let $G'_< = \partial(t)$. We state that*

$$\forall \widehat{G}'_< \in \texttt{gen}(G'_<, t),\ \texttt{rem}^*(\widehat{G}'_<, t) = G'_<$$

Proof. In accordance to Definition 4.13, $\widehat{G}'_<$ is generated by the `gen` operation either directly on $G'_<$, using one of the $\texttt{gen}_{P_i}$ operations, or on some nested subgraph of $G'_<$.

However, even though $\texttt{gen}_{P_1}$, $\texttt{gen}_{P_2}$, and $\texttt{gen}_{P_3}$ directly modify the sequence graph they operate on, $\texttt{gen}_{P_4}$ acts recursively on the branches of the conditional structures of the graph. Thus, the proof is carried on by structural induction on the nesting level at which the insertion is located, with $\texttt{gen}_{P_1}$, $\texttt{gen}_{P_2}$, and $\texttt{gen}_{P_3}$ representing the base cases, and the recursive calls in $\texttt{gen}$ and $\texttt{gen}_{P_4}$ providing the induction steps.

As base case, let us consider the case in which the insertion occurs directly on some graph $\tilde{G}_< = (\tilde{V}_<, \tilde{E}_<, \tilde{\Sigma}_<)$ using one of the patterns 1, 2 or 3. Let us consider each of them separately.

— *Pattern 1.* There are two cases for $\texttt{gen}_{P_1}$.

If $\tilde{G}_< = (\emptyset, \emptyset)$ then $\texttt{gen}_{P_1}(\tilde{G}_<, t) = \{(\{\{t\}\}, \emptyset)\}$. By removing the task, we obtain $\texttt{rem}((\{\{t\}\}, \emptyset), t) = (\{\emptyset\}, \emptyset)$. Since the node containing t is now empty, the normalisation rule $\texttt{norm}_1$ generates an empty graph, i.e., $\texttt{norm}_1((\{\emptyset\}, \emptyset)) = (\{\}, \emptyset) = (\emptyset, \emptyset)$, as desired. As $(\emptyset, \emptyset)$ is the main graph, it is preserved by the normalisation rules. Thus, for all $\tilde{G}'_< \in \texttt{gen}_{P_1}(\tilde{G}_<, t),\ \texttt{rem}(\tilde{G}'_<, t) = \tilde{G}_<$.

Else, if $\tilde{G}_<$ is not empty, Pattern 1 inserts t in a new node $\{t\}$ that is added at the beginning, the end, or between any two nodes of $\tilde{G}_<$, creating a new graph $\tilde{G}'$. Thus, removing t from $\tilde{G}'_<$, that is, $\texttt{rem}(\tilde{G}'_<, t)$, actually removes t from $\{t\}$ as it is its closest node. This leads to an empty node $\{\}$, which is deleted by the normalisation rule $\texttt{norm}_1$. Thus, for all $\tilde{G}'_< \in \texttt{gen}_{P_1}(\tilde{G}_<, t),\ \texttt{rem}(\tilde{G}'_<, t) = \tilde{G}_<$.

— *Pattern 2.* In the definition of $\texttt{gen}_{P_2}$, the task t is inserted in a node next to a subsequence of nodes of $\tilde{G}_<$, called $\tilde{\tilde{G}}_< \in \mathcal{P}^*_O(\tilde{G}_<)$. To do so, the subsequence is replaced by a node containing only t and the subsequence, now behaving as an independent sequence graph. The generated graph is called $\tilde{G}'_<$. Removing t from $\tilde{G}'_<$ corresponds to removing t from the node $\{t, \tilde{\tilde{G}}_<\}$, which leaves it containing only $\tilde{\tilde{G}}_<$. As this node now contains only a single sequence graph, which is by construction not a task boundary nor the main graph, it is eligible for being removed by the normalisation rule $\texttt{norm}_2$, which consequently leaves $\tilde{G}'_<$ in the same disposition than $\tilde{G}_<$. Thus, for all $\tilde{G}'_< \in \texttt{gen}_{P_2}(\tilde{G}_<, t),\ \texttt{rem}(\tilde{G}'_<, t) = \tilde{G}_<$.

— *Pattern 3.* In the definition of $\texttt{gen}_{P_3}$, the task t is inserted before or after any combination c of elements of any node $\tilde{v}_<$ of $\tilde{G}_<$. This is done by creating a sequence graph $G^n_<$ composed of two nodes, one containing only t, that is, $\{t\}$, and one containing only the combination, that is, $\{c\}$, and an edge which is either $\{c\} \to \{t\}$, or $\{t\} \to \{c\}$. Thus, we have either $G^n_< = (\{\{t\}, \{c\}\}, \{\{t\} \to \{c\}\}, \Sigma_<)$ or $G^n_< = (\{\{t\}, \{c\}\}, \{\{c\} \to \{t\}\}, \Sigma_<)$. $G^n_<$ then replaces the original combination of elements in $\tilde{v}_<$. This leads to a new sequence graph $\tilde{G}'_<$. By removing t from $\tilde{G}'_<$, $G^n_<$ now looks like $(\{\{c\}, \{\}\}, \{\{c\} \to \{\}\}, \Sigma_<)$ or $(\{\{c\}, \{\}\}, \{\{\} \to \{c\}\}, \Sigma_<)$. The empty node of $G^n_<$ is removed by the normalisation rule $\texttt{norm}_1$, which leads to a graph $G^{n'}_< = (\{\{c\}\}, \emptyset, \Sigma_<)$. Thus, $\tilde{G}'_<$ now contains a node itself containing a sequence graph with only one node. According to the definition of the normalisation

rule $\texttt{norm}_3$, this sequence graph is useless and the elements of its unique node can be reinserted in the node hosting this sequence graph. Consequently, the elements of c are added back to $\tilde{v}_<$. Thus, for all $\tilde{G}'_< \in \texttt{gen}_{P_3}(\tilde{G}_<, t),\ \texttt{rem}(\tilde{G}'_<, t) = \tilde{G}_<$.

Once proved for the base case, let us assume that the property holds for any sequence graph of nesting level n. Two cases must then be considered for the induction step.

The first case considers Pattern 4, which acts on choice structures in $\tilde{G}_<$. Specifically, given a node $\tilde{v}_< \in \tilde{V}_<$, and given a conditional structure $s = \{(\tilde{G}^1_<, p_1), ..., (\tilde{G}^n_<, p_n)\}$ in $\tilde{v}_<$, the set of sequence graphs generated by operation $\texttt{gen}_{P_4}$ is

$$\texttt{gen}_{P_4}(\tilde{G}_<, t) = \bigcup_{(\widehat{\tilde{G}}^1_<, ..., \widehat{\tilde{G}}^n_<) \in \prod_{i=1}^{n} \texttt{gen}(\tilde{G}^i_<, t)} \{(\tilde{V}'_<, \tilde{E}_<, \tilde{\Sigma}_< \cup \{\sigma(t)\})\}$$

where $\tilde{V}'_< = \tilde{V}_< \setminus \{\tilde{v}_<\} \cup \{\tilde{v}_< \setminus \{s\} \cup \{((\widehat{\tilde{G}}^1_<, p_1), ..., (\widehat{\tilde{G}}^n_<, p_n))\}$. By induction hypothesis, for all $i \in [1...n]$, for all $\widehat{\tilde{G}}^i_< \in \texttt{gen}(\tilde{G}^i_<, t),\ \texttt{rem}^*(\widehat{\tilde{G}}^i_<, t) = \tilde{G}^i_<$ because the $\tilde{G}^i_<$ have a nested level equal to 1 (they are the topmost sequence graphs on which the gen function is called). As the generated choice structure simply replaces the old one, the property also holds on $\tilde{G}_<$. As the property holds on $\tilde{G}_<$, it holds on a graph having a nesting level of $n + 1$. Thus, for all $\widehat{\tilde{G}}_< \in \texttt{gen}_{P_4}(\tilde{G}_<, t)$, $\texttt{rem}^*(\widehat{\tilde{G}}_<, t) = \tilde{G}_<$.

The second case takes into account the recursive call of the gen function on all subgraphs of $\tilde{G}_<$, that are, the $\tilde{v}' \in S_{\tilde{G}_<} = \bigcup_{\tilde{v}_< \in \tilde{V}_<} \{\tilde{v} \in \tilde{v}_< \mid \theta(\tilde{v}) = \mathcal{G}\}$. With regards to the current gen call, all these subgraphs have a nested level of 1. Thus, the induction hypothesis holds on all these graphs, i.e., for all $\tilde{v}' \in S_{\tilde{G}_<}$, for all $\overline{G}_< \in \texttt{gen}(\tilde{v}', t),\ \texttt{rem}^*(\overline{G}_<, t) = \tilde{v}'$. As the original graph $\tilde{v}'$ is simply replaced by the generated one $\overline{G}_<$, the property also holds on the original graph $\tilde{G}_<$. As the property holds on $\tilde{G}_<$, it holds on a graph having a nesting level of $n + 1$. Thus, for all $\widehat{\tilde{G}}_< \in \texttt{gen}(\tilde{G}_<, t)$, $\texttt{rem}^*(\widehat{\tilde{G}}_<, t) = \tilde{G}_<$. □

Arising from the previous proof, the result on the boundary of the removed task may be extended to the entire sequence graph.

Corollary 4.1 (Structure Persistency)**.** *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, and let $t \in \mathcal{T}(G_<)$ be a task of $G_<$. We state that*

$$\forall G'_< \in \texttt{ins}(\texttt{rem}(G_<, t), t),\ \texttt{rem}^*(G'_<, t) = \texttt{rem}(G_<, t)$$

Proof. Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, and let $t \in \mathcal{T}(G_<)$ be a task of $G_<$. By Definition 4.12, $\texttt{ins}(\texttt{rem}(G_<, t), t)$ returns the set of sequence graphs generated by replacing $\partial(t)$ by one of the graphs belonging to $S_{\widehat{G_<}} = \texttt{gen}(\texttt{rem}(\partial(t), t), t)$ in $G_<$. Moreover, by Proposition 4.2, for each $\widehat{G}_< \in S_G$, $\texttt{rem}^*(\widehat{G}_<, t) = \texttt{rem}(\partial(t), t)$. But then, by

Definition 4.10 and Remark 4.2, we can conclude that for each $G'_<$ in $\texttt{ins}(\texttt{rem}(G_<, t), t)$, $\texttt{rem}^*(G'_<, t) = \texttt{rem}(G_<, t)$. □

If the structure of the graph without considering the moved task is the same, then its traces must also be the same.

Corollary 4.2. *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, and let $t \in \mathcal{T}(G_<)$ be a task of $G_<$. We state that for all $G'_< \in \texttt{ins}(\texttt{rem}(G_<, t), t)$:*

— $\forall \lambda_< \in \Lambda(G_<),\ \exists \lambda'_< \in \Lambda(G'_<)$ *such that* $\lambda_< \setminus \{t\} = \lambda'_< \setminus \{t\}$;
— $\forall \lambda'_< \in \Lambda(G'_<),\ \exists \lambda_< \in \Lambda(G_<)$ *such that* $\lambda'_< \setminus \{t\} = \lambda_< \setminus \{t\}$.

Proof. Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, and let $t \in \mathcal{T}(G_<)$ be a task of $G_<$. Since, by Corollary 4.1, for all $G'_< \in \texttt{ins}(\texttt{rem}(G_<, t), t)$, $\texttt{rem}^*(G'_<, t) = \texttt{rem}(G_<, t)$, that is, $G_<$ and $G'_<$ are identical after removing t, then they must have the exact same traces without t. □

Moreover, there is not only a correspondence between the execution traces of the original graph deprived of one of its tasks and the ones of any graph obtained by removing and inserting back this task also deprived from it. Indeed, as both graphs have the exact same structure, their execution traces must be strictly identical.

Corollary 4.3. *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, and let $t \in \mathcal{T}(G_<)$ be a task of $G_<$. We state that*

$$\forall G'_< \in \texttt{ins}(\texttt{rem}(G_<, t), t),\ \Lambda(\texttt{rem}^*(G'_<, t)) = \Lambda(\texttt{rem}(G_<, t))$$

Proof. The proof follows directly from Proposition 4.2 and Corollaries 4.1 & 4.2. □

This results can be expressed in a different form.

Corollary 4.4. *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, and let $t \in \mathcal{T}(G_<)$ be a task of $G_<$. We state that*

$$\forall G'_< \in \texttt{ins}(\texttt{rem}(G_<, t), t),\quad \bigcup_{\lambda \in \Lambda(G_<)} \{\lambda \setminus \{t\}\} = \bigcup_{\lambda' \in \Lambda(G'_<)} \{\lambda' \setminus \{t\}\}$$

Proof. The proof follows directly from Proposition 4.2 and Corollaries 4.1, 4.2, & 4.3. □

The above results show that the structure, and hence the execution traces of the graph, are preserved by the movement of any of its tasks, without considering it. Furthermore, by definition of the patterns responsible of moving a task, we can guarantee that the number of times a moved task occur remains the same as in the original graph. Said differently, this means that the number of occurrences of a task in a trace of the original sequence graph remains identical throughout the lifetime of this graph. This implies that not only

there is a correspondence between the execution traces of a graph and another where a task has been moved, but one of the traces of the original graph has to be a permutation of a trace of the resulting graph, and vice-versa.

Proposition 4.3. *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, and let $t \in \mathcal{T}(G_<)$ be a task of $G_<$. We state that*

— $\forall G'_< \in \texttt{ins}(\texttt{rem}(G_<, t), t),\ \forall \lambda_< \in \Lambda(G_<),\ \exists \lambda'_< \in \mathfrak{S}(\Lambda(G'_<)) \mid \lambda_< = \lambda'_<$;
— $\forall G'_< \in \texttt{ins}(\texttt{rem}(G_<, t), t),\ \forall \lambda'_< \in \Lambda(G'_<),\ \exists \lambda_< \in \mathfrak{S}(\Lambda(G_<)) \mid \lambda'_< = \lambda_<$.

Proof. Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, and let $t \in \mathcal{T}(G_<)$ be a task of $G_<$. The existence of a trace $\lambda_< \in \Lambda(G_<)$ and a trace $\lambda'_< \in \Lambda(G'_<)$ such that $\lambda_< \setminus \{t\} = \lambda'_< \setminus \{t\}$ is already stated in Corollary 4.2. The fact that $\lambda_<$ is a permutation of $\lambda'_<$ and oppositely basically describes that the number of occurrences of t is identical in both $\lambda_<$ and $\lambda'_<$.

By definition, the patterns used to insert a task back into a graph inherently preserve the number of occurrences of a task. Indeed, these patterns are applied on the boundary of t, which is by definition the closest conditional structure surrounding t. Thus, not exceeding it ensures that the conditionality of t remains the same throughout the refactoring steps.

Regarding the fourth pattern, which may duplicate a task and thus insert it several times, the key is to remember that when a task is duplicated, these duplicates are inserted in all the branches of the choice targeted by the pattern. By doing so, this task remains exactly as conditional as it was in the original version of the graph, because any branch of the choice will execute it. Consequently, the number of occurrences of this task does not vary by applying Pattern 4. □

4.4 Task Dependencies

In business processes, tasks are naturally ordered by the sequence flows that connect them. Thus, two tasks connected by a sequence flow are dependent, as one must be executed before the other. As the solution presented in this chapter performs a restructuring of the process, there is no guarantee regarding the final position of a task in the resulting process, compared to its position in the original one. Nonetheless, some tasks may have to remain in a specific order to preserve the meaning of the process (e.g., some product should be collected before being packaged, or packaged before being delivered). In the refactoring procedure, dependencies between tasks may change. However, a set of *strong dependencies* will be guaranteed to remain. Such strong dependencies will be assumed from the beginning of the refactoring procedure, either given by the user or computed by analysing the data-flow graph corresponding to the business process [EG16, DRS18c]. They are similar to the order imposed by the sequential operator '$<$' appearing in the constraints presented in Chapter 3.

Definition 4.20 (Task Dependency)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process. A*

dependency *between two tasks* $t_i, t_j \in V$ *is represented as* $t_i \prec t_j$*, or* (t_i, t_j)*. The set of task dependencies of* G *is noted* $\texttt{Dep}$*.*

Given tasks t_i and t_j, such that $(t_i, t_j) \in \texttt{Dep}$, we say that t_j depends on t_i. We say that two tasks t_i and t_j are *dependent* if (t_i, t_j) or $(t_j, t_i) \in \texttt{Dep}$, and *independent* otherwise. If $t_i \prec t_j$, then t_i is said to be a *predecessor* of t_j, and t_j a *successor* of t_i.

Definition 4.21 (Dependency Satisfaction)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process and let* $t_i, t_j \in V$ *be two tasks such that* $(t_i, t_j) \in \texttt{Dep}$*. We say that dependency* (t_i, t_j) *is* satisfied *in* G *if and only if there does not exist* $\lambda \in \Lambda(G)$ *such that* $(t_i \in \lambda) \wedge (t_j \in \lambda) \wedge (\texttt{index}(t_j) < \texttt{index}(t_i))$*.*

This notion can be extended at the level of the BPMN process to the whole set of dependencies.

Definition 4.22 (Dependencies Satisfaction)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process and let* $\texttt{Dep}$ *be its set of dependencies.* G *is said to* satisfy the dependencies*, noted* $G \models \texttt{Dep}$*, if and only if for all* $(t_i, t_j) \in \texttt{Dep}$*,* (t_i, t_j) *is satisfied in* G*.*

Example. The running example displayed in Figure 4.1 contains several dependencies that should be preserved by the refactoring operations. In particular, we have $\texttt{Dep} = \{$(Collect goods, Prepare parcel), (Send payment request, Check payment reception), (Check payment reception, Validate payment), (Validate payment, Deliver by drone), (Validate payment, Deliver by bike), (Prepare parcel, Deliver by drone), (Prepare parcel, Deliver by bike), (Fill in admin. doc., Deliver by drone), (Fill in admin. doc., Deliver by bike)$\}$. As an example, dependency (Send payment request, Check payment reception) indicates that task Check payment reception can not execute before task Send payment request. However, a dependency such as (Log information, Validate payment) is unnecessary and does not exist, because both tasks can execute without any constraint regarding their order.

4.5 Fixed Durations Approach

The first approach that we propose performs a "one-shot" refactoring of the BPMN process G given as input, in the sense that the process is modified once in a monolithic way. This approach only supports processes having fixed durations for tasks and IAT, that are, durations following a constant distribution. The idea of this approach is to first build a new BPMN process G' having the shortest stochastic worst-case execution time in a infinite resources context, while satisfying the dependencies of G. Then, the minimum pool of resources needed by this optimal process to execute without latencies is computed. If this pool is lower than or equal to the real pool of resources that G' has access to, then G' is considered optimal, and returned to the user. Otherwise, several specific tasks responsible of the overuse of some of the resources are removed from their parallel structures and put in sequence, in order to reduce the AET of the process by lowering its synchronisation times. The whole approach is summarised in Figure 4.12. This approach was implemented

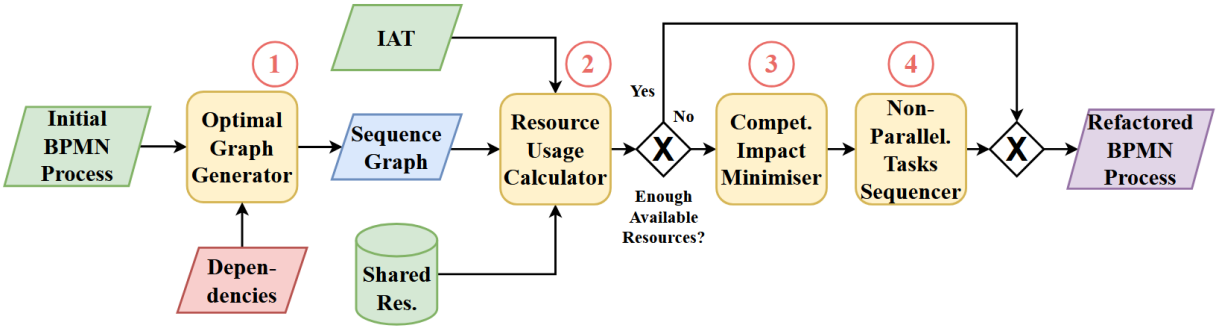


Figure 4.12: Overview of the Approach

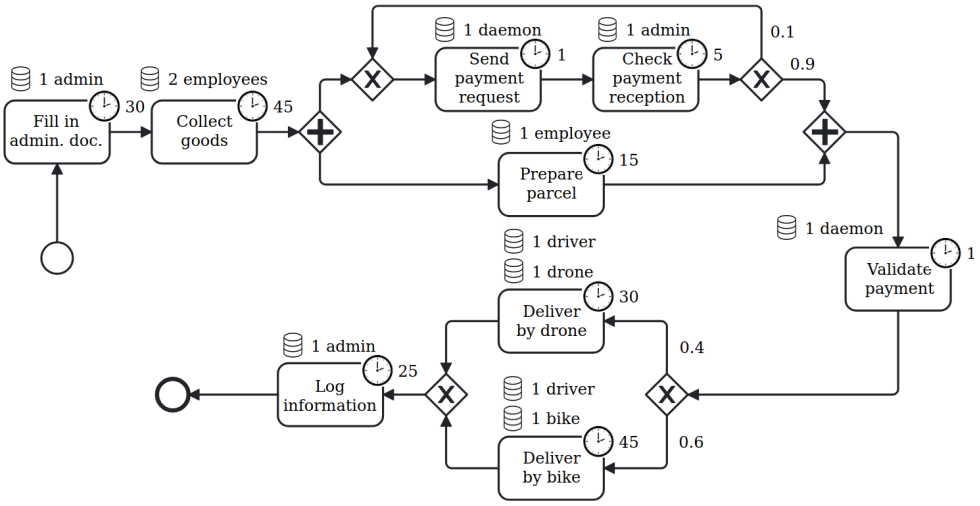


Figure 4.13: Running Example with Fixed Durations

as a tool that was used for test and validation purposes. It is detailed in Section 5.3.1.

Running example. The running example used throughout this section is a version of the BPMN process shown in Figure 4.1 where tasks have been assigned constant durations, as required by this approach. It is given in Figure 4.13. Similarly, the inter-arrival time of the instances of the process is set to a constant value of 25 minutes. Considering the durations of the tasks of the process, a single instance of it should take at least 146m (2h26m) to complete, depending on the number of times the payment will not be received, or the mode of delivery of the good. When all the instances are running together, the average time taken by an instance to complete goes up to 767m (12h47m).

4.5.1 Generation of the Optimal Sequence Graph

The first step of this approach consists in computing the *optimal* version of the BPMN process given as input. For now, what we consider an optimal BPMN process is a process obtaining the shortest stochastic worst-case execution time when executed a single time

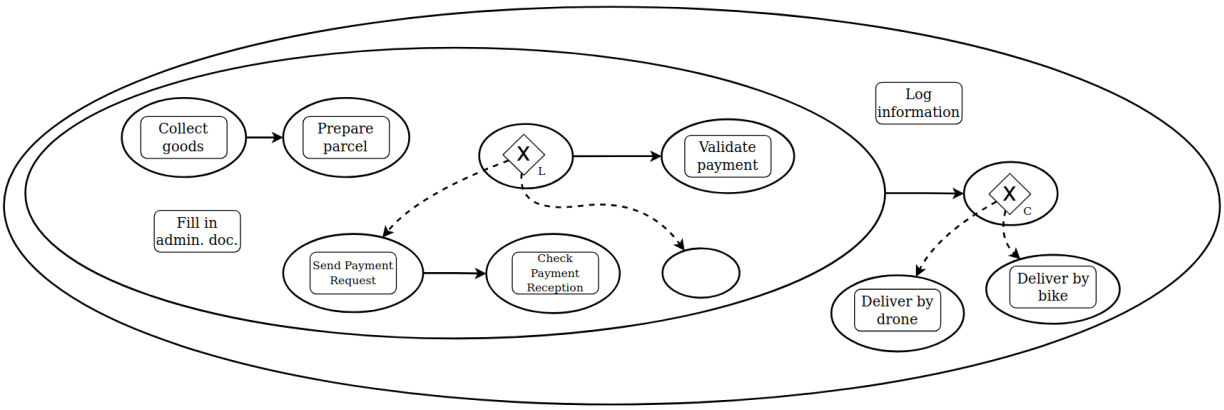


Figure 4.14: Optimal Version of the Sequence Graph Representation of the Running Example

in an infinite resources context. Under these assumptions, the optimal version of the original process is a fully parallel process, where every tasks and conditional structures are in parallel. The generation of this optimal process is simple, although it has to preserve the eventual dependencies of the process. Indeed, it suffices to put the tasks belonging to `Dep` into sequence nodes, connect them, and put every new sequence graphs in parallel. However, as shown in Section 4.3, to preserve the trace semantics of a sequence graph, one must not move a task outside of its boundary. To do so, the dependencies in `Dep` are temporarily rewritten as dependencies between their boundaries, if any. According to the formalism of Section 4.3, it is also allowed to put tasks inside every sequence graph belonging to a choice. For this reason, there might be several semantically equivalent but syntactically different versions of the optimal sequence graph. The version that is kept is the one in which the fewest task duplications have been performed. Indeed, duplicating a task and inserting it inside a choice structure has the drawback of reducing its moving freedom, as its boundary is now the choice structure in which it has been inserted.

Example. Considering the sequence graph of the running example along with its dependencies, one can build the optimal sequence graph shown in Figure 4.14. In order to preserve the trace semantics of the original sequence graph, the reader can see that, for instance, task Validate payment has been put after $\times_L$, and not directly after task Check payment reception inside $\times_L$, which would have induced multiple occurrences of task Validate payment in several traces, unlike in the original process.

4.5.2 Computation of Resource Usage

In the previous step, we have built a sequence graph that is an optimal representation of our initial process, and which satisfies the dependencies provided by the user. Nonetheless, this graph is optimal under the assumptions that it is executed a single time in an unlimited resources context. In practice, the pool of resources is usually limited. For instance,

a company will have a finite number of employees, or machines, and will have to deal with them. In a limited resources context, this sequence graph would also be an optimal representation of the initial process if it was executed a single time, i.e., in a single instance context. However, as long as several instances of the process are involved, this sequence graph may no longer be optimal. Indeed, if some resources are not sufficiently represented, the execution of the process may be delayed, due to eventual additional synchronisation times coming from the parallel merge gateways.

The goal of this step is to verify whether our current version of the process can be executed without involving such overheads. To do so, our proposal consists in statically computing the pool of resources that would be needed by the process to execute without any competition for the resources, called $P_{=}$. If the available pool of resources of the process P is smaller than $P_{=}$ (i.e., $P \subseteq P_{=}$), we consider that the current sequence graph is optimal. The corresponding BPMN process is then synthesised and returned to the user as optimal version of the original process. Otherwise, it might be the case that a less parallel version of this graph would obtain a lower average execution time. To avoid such an overhead, steps 3 and 4 of Figure 4.12 are performed, with the goal of sequencing some tasks which would potentially induce delays if they remain in parallel. In this case, the BPMN process returned to the user is an optimised version of the original one, but provides no guarantee of optimality.

To compute the aforementioned pool of resources, the idea is first to statically compute the tasks that may be executing at any given time of the execution of a single instance of the process. These tasks are not representing an execution of the process, but depicts all the possible combinations of tasks currently executed at any time of the (real) execution of the process. This is particularly useful in case of conditional structures (choices and loops), as a single execution of the process would have executed a single branch of a choice structure, or the loop a certain number of times but not necessarily enough times. In an infinite resources context, each task of the process can execute as soon as it is ready to. Moreover, the durations of the tasks are, in this approach, following a constant distribution. Under these assumptions, this set of tasks can be computed using the stochastic worst-case execution time of a sequence graph, itself computable statically.

Definition 4.23 ((Statically Computed) Stochastic Worst-Case Execution Time)**.** *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph and P_∞ be an infinite resource pool. In a single instance context, where the durations of the tasks of $G_<$ follow a constant distribution, the* stochastic worst-case execution time *of $G_<$ is (recursively) defined as*

$$SWCET(G_<, P_\infty) = \sum_{v_< \in V_<} SWCET(v_<, P_\infty)$$

where

$$SWCET(v_<, P_\infty) = \max_{v \in v_<} SWCET(v, P_\infty)$$

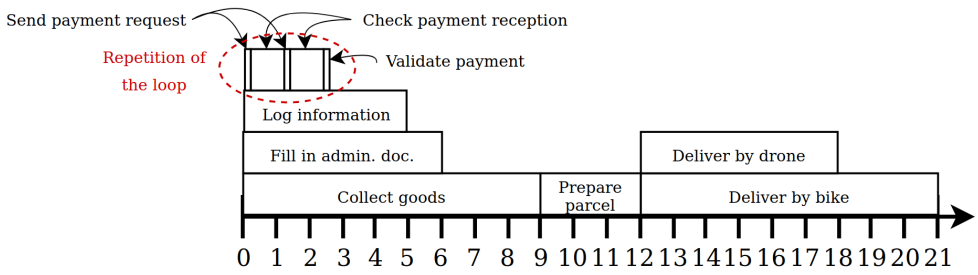


Figure 4.15: Visual Representation of the Execution Flow of the Running Example

where

$$SWCET(v, P_\infty) = \begin{cases} SWCET(v, P_\infty) & \text{if } \theta(v) = \mathcal{G} \\ \max\limits_{(G^i_<, p_i) \in v} \{SWCET(G^i_<, P_\infty)\} & \text{if } \theta(v) = \mathcal{C} \\ (i+1) \times SWCET(G_{FL}, P_\infty) + i \times SWCET(G_{LF}, P_\infty) & \text{if } \theta(v) = \mathcal{L} \\ \delta(v) & \text{if } \theta(v) = \mathcal{T} \end{cases}$$

where $i \in \mathbb{N}$ *such that* $(p_{LF})^i \leq 0.01$ *and* $(p_{LF})^{i-1} > 0.01$.

Example. Given the above formulae and the optimal version of the sequence graph of the running example, we can compute its stochastic worst-case execution time. Here, it corresponds to the maximum between the duration of task Log information and the duration of the sequence graph containing all the remaining tasks. This sequence graph's duration is itself the addition of (i) the maximum between the duration of task Fill in admin. doc., the duration of the sequence graph containing tasks Collect goods and Prepare parcel in sequence, and the duration of the sequence graph containing the loop structure executed two times (after which its probability becomes equal to the threshold of 0.01) and the task Validate payment in sequence, and (ii) the duration of the choice structure. This gives a stochastic worst-case execution time of 105m, or 1h45. Figure 4.15 provides a visual representation of this computation, where 5 minutes correspond to a space of 1 for the sake of space.

Based on these definitions, the set of tasks potentially executing at the same time in the worst case, called $\tilde{S_T}$, maps every value in $[0...SWCET(G_<, P_\infty)]$ to the tasks potentially executing at this instant of time, retrievable with the operation $\tilde{S_T}(t)$.

Example. Considering the running example, the set of tasks potentially executing at the same time in the worst case is $\tilde{S_T} =$

- $[0...1[\rightarrow$ {Send payment request, Log information, Fill in admin. doc., Collect goods}
- $[1...6[\rightarrow$ {Check payment reception, Log information, Fill in admin. doc., Collect goods}
- $[6...7[\rightarrow \tilde{S_T}(0)$
- $[7...12[\rightarrow \tilde{S_T}(1)$

— $[12...13[\rightarrow$ {Validate payment, Log information, Fill in admin. doc., Collect goods}
— $[13...25[\rightarrow$ {Log information, Fill in admin. doc., Collect goods}
— $[25...30[\rightarrow$ {Fill in admin. doc., Collect goods}
— $[30...45[\rightarrow$ {Collect goods}
— $[45...60[\rightarrow$ {Prepare Parcel}
— $[60...90[\rightarrow$ {Deliver by drone, Deliver by bike}
— $[90...105[\rightarrow$ {Deliver by bike}

The notion of *potential execution* of tasks is clearer here, where, for instance, $\tilde{S_T}(65)$ contains both tasks Deliver by drone and Deliver by bike, although only one of them will be executed by the process.

However, in this work, we are interested not only in executing a single instance of a process, but several of them. As the IAT follows a constant distribution, the *maximum number of instances executing simultaneously* is computable statically.

Definition 4.24 (Maximum Number of Simultaneous Instances)**.** *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph such that for all $v \in \mathcal{T}(G_<)$, $\delta(v) \in \mathbb{N}$, let $IAT \in \mathbb{N}$ be the inter-arrival time of $G_<$. The* maximum number of simultaneous instances *of $G_<$ is defined as*

$$I_{max} \stackrel{\text{def}}{=} 2 \times I_{B/A} - 1$$

where

$$I_{B/A} \stackrel{\text{def}}{=} \lceil \frac{SWCET(G_<, P_\infty)}{IAT} \rceil$$

The quantity $I_{B/A}$ represents both the number of instances that were already running before the start of the considered instance (including it), and the number of instances that will start during its execution (including it). It is consequently doubled to consider both the instances before and the ones after the main one, and lowered by 1 in order to remove the main instance counted twice. More than simply being able to compute the maximum number of simultaneous instances, we also know the shift between the execution of the running instances, which is the IAT. Thus, we know that when an instance I is at time t of its execution, the p^{th} instance that was already running before the start of I is at time $t - p \times IAT$ of its execution, while the n^{th} instance that started running after I is at time $t + n \times IAT$ of its execution. Given this information, we can compute the *stochastically optimal pool of resources* of our process, by retrieving the maximum usage of each resource throughout the execution of multiple instances of the process.

Definition 4.25 (Stochastically Optimal Pool of Resources)**.** *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, let S_{res} be the set of resources used by $G_<$, let $\tilde{S_T}$ be the set of tasks executing at the same time in the worst-case, let IAT be the inter-arrival time of the*

instances of $G_<$. We define the stochastically optimal pool of resources *of $G_<$ as*

$$P_= \stackrel{\text{def}}{=} \bigcup_{r \in S_{res}} \{r \to \lceil \max_{t \in [0 \ldots SWCET(G_<, P_\infty)]} \sum_{i=-I_{B/A}+1}^{I_{B/A}-1} \sum_{v \in \tilde{S_T}(i \times IAT + t)} p(v) \times \rho(v)(r) \rceil\}$$

where i represents the index of the considered instance to ensure a proper shift of the IAT.

This pool of resources is then eventually compared to the available one. Basically, if $P \subseteq P_=$, the available pool of resources contains a sufficient number of replicas of each resource to execute an arbitrarily large number of instances of our process without introducing significant delays in it. However, if $P \nsubseteq P_=$, then some tasks of the process may compete to acquire the resources they need, which will lead to latencies in the execution of the process, and thus to a greater AET. The next step consequently consists in minimising the competition of such resources.

Example. Considering the set of potentially executing tasks of the running example, its IAT, and its stochastic worst-case execution time, the stochastically optimal pool of resources is $P_= = \{\mathsf{driver} \to 2, \mathsf{drone} \to 1, \mathsf{bike} \to 2, \mathsf{employee} \to 5, \mathsf{admin} \to 4, \mathsf{daemon} \to 1\}$. This computation is made more visual by representing the execution flows of each instance of the process running in parallel, that is, an instance called *main instance* (Figure 4.15), 4 instances already running when the main instance started (Figure 4.16 (a)), and 4 instances that started running after the main instance (Figure 4.16 (b)). Reminding that $P = \{\mathsf{driver} \to 2, \mathsf{drone} \to 2, \mathsf{bike} \to 2, \mathsf{employee} \to 6, \mathsf{admin} \to 2, \mathsf{daemon} \to 2\}$, we see that $P \nsubseteq P_=$, because there are not enough replicas of resource admin in P. Thus, the impact of the competition for this resource has to be quantified.

4.5.3 Quantification & Minimisation of the Resource Competition Impact

At this stage of the approach, we remarked that the stochastically optimal resource pool of our sequence graph $P_=$ was smaller than its available resource pool P. As a consequence, some tasks of the process will not be able to execute as soon as they should, because their corresponding resources will not be available at that time. The goal of this step is to determine whether this shift in the execution of certain tasks will induce long synchronisation times in the process, and thus if putting these tasks in sequence would allow to obtain a shorter AET than if these tasks remained in parallel. Thanks to the previous computations, we know precisely the number of resources required by the process at any given time of its execution. Based on these information, we compute a value called *absorbance*, which reflects the capacity of the process to absorb the lack of certain resources without causing a significant increase of its AET. It is defined as the ratio between the amount of time during which the lacking resources are overused and the amount of time during which they are underused.

Definition 4.26 (Absorbance)**.** *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, let $\tilde{S_T}$ be the*

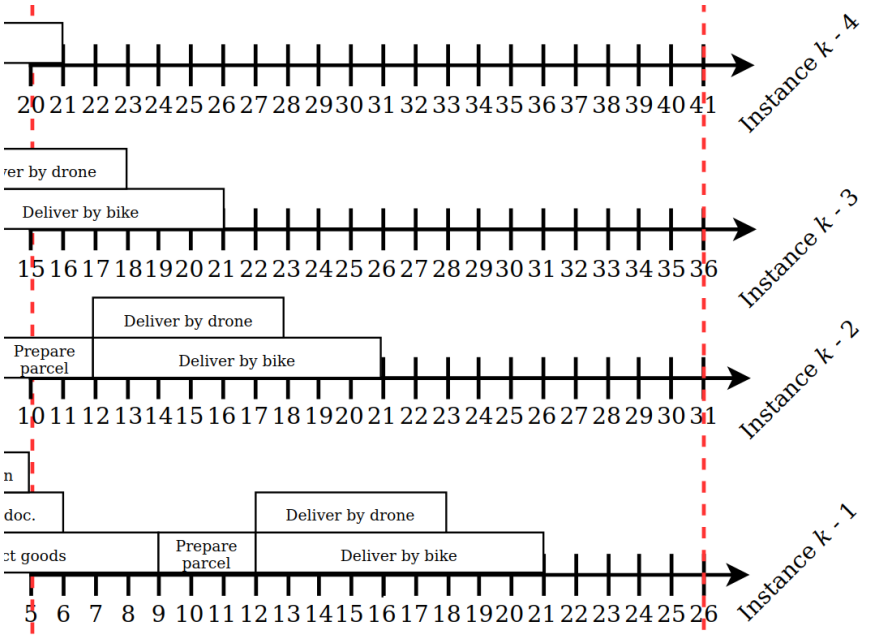


Figure 4.16: (a) Tasks Execution Flows of the Already Running Instances

set of tasks executing at any given time of the execution of $G_<$*, let* P *be its available pool of resources and* $P_=$ *be its stochastically optimal one. For all* r *in* P *such that* $P(r) > P_=(r)$*, we define the* absorbance *of* r *as*

$$\texttt{absorbance}(r) \stackrel{\text{def}}{=} \frac{\left| \bigcup_{t \in [0 \ldots \mathit{SWCET}(G, P_\infty)]} \{t \mid \sum_{-I_{A/B}+1}^{I_{A/B}-1} \sum_{v \in \tilde{S_T}(i \times IAT + t)} p(v) \times \rho(v)(r) > P(r)\} \right|}{\left| \bigcup_{t \in [0 \ldots \mathit{SWCET}(G, P_\infty)]} \{t \mid \sum_{-I_{A/B}+1}^{I_{A/B}-1} \sum_{v \in \tilde{S_T}(i \times IAT + t)} p(v) \times \rho(v)(r) < P(r)\} \right|}$$

If this value is below a certain threshold (100 by default, based on our experiments), we conclude that the lack of the incriminated resource will not result in a significant increase of the AET of the process (i.e., the process will *absorb* the induced delays). If this is the case for each lacking resource, the process does not need any modification, and is consequently returned as is to the user. Otherwise, if some lacking resources have an absorbance that is above the threshold, we consider that some tasks have to be removed from their parallel structures and put in sequence to prevent the appearance of significant synchronisation times in the process.

Example. According to the available pool of resources of the running example P and the stochastically optimal one $P_=$, we can see that we lack some replicas of resource admin. Its usage during the execution of the process is given in Figure 4.17. As the reader can see, it is overused most of the time, and never underused. This means that it has an absorbance of ∞ (the denominator of the fraction is equal to 0), and consequently that its overuse

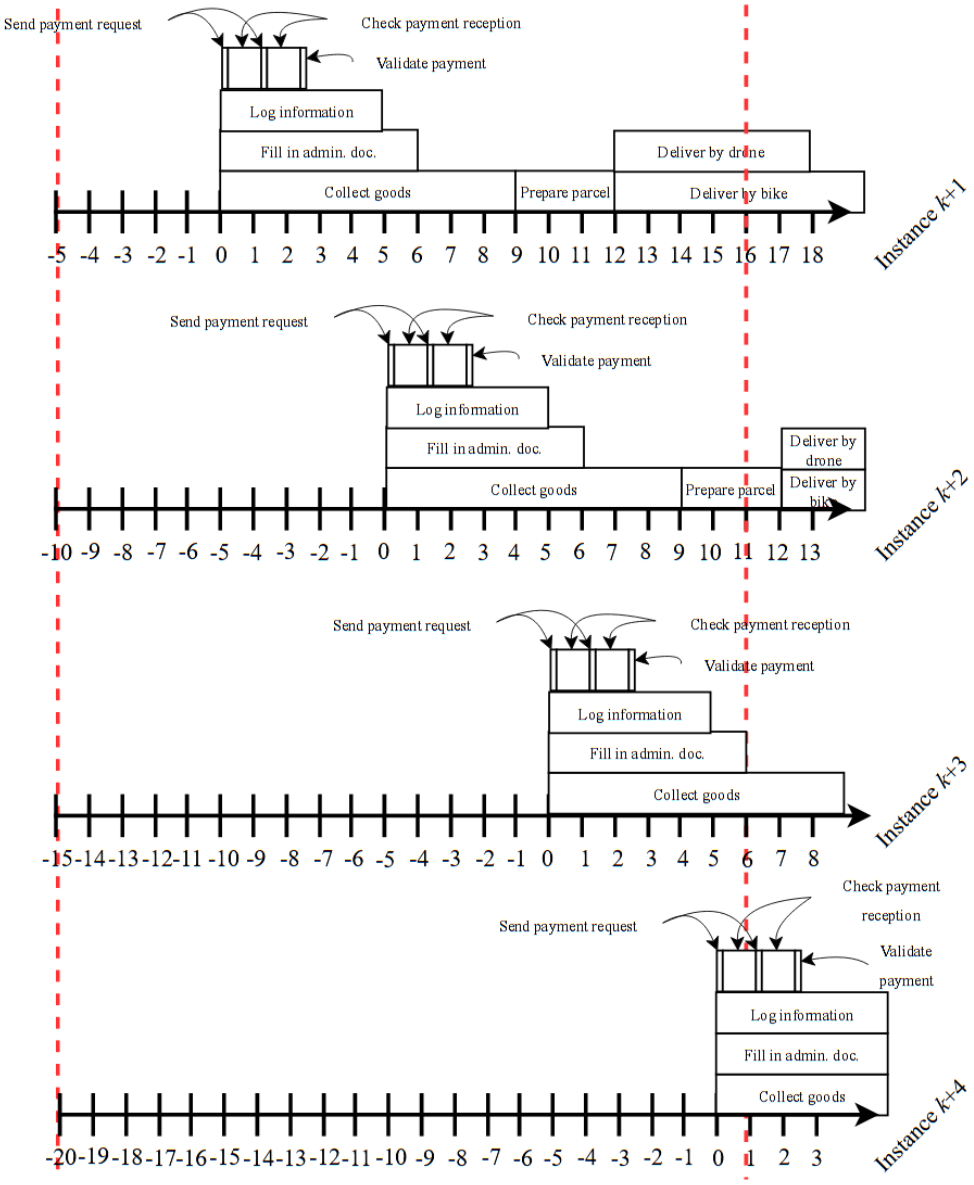


Figure 4.16: (b) Tasks Execution Flows of the Instances that Started after the Main One

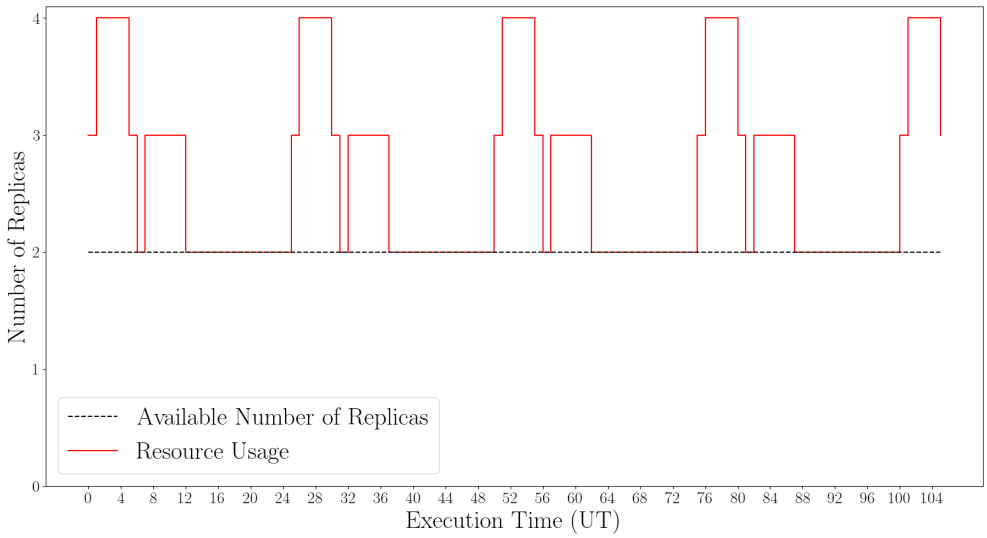


Figure 4.17: Usage of Resource admin Over Time

cannot be handled properly by the process, thus resulting in additional delays or latencies in the execution of the process. The result of this analysis is that some of the tasks making use of resource admin have to be sequenced.

The idea of this sequencing is to proceed step by step in order to keep as many tasks as possible in parallel, while reaching an absorbance lower than the threshold. To do so, the tasks are removed one by one from their parallel structures, and the absorbance of the new process is recomputed after each sequencing. However, the order in which the tasks are selected for sequencing matters. Indeed, a task having an important duration has more impact on the AET of the process, and should thus probably have a higher probability of remaining in parallel than a task having a short duration. Oppositely, a task requiring many replicas of a given lacking resource is more likely to provoke synchronisation delays than a task requiring a fewer number of replicas of this resource. Thus, it should have a lower probability of remaining in parallel than its sibling. Consequently, the tasks requiring a lacking resource subject to sequencing are ordered by a *score* representing how impactful it is to sequence them. The higher the score, the higher the interest of being sequenced.

Definition 4.27 (Score of a Task)**.** *Let $G_< = (V_<, E_<, \Sigma_<)$ be a sequence graph, and let P be the available pool of resources of G. For all $t \in \mathcal{T}(G_<)$, we define the* score *of t with regards to r as*

$$\texttt{score}(t, r) \stackrel{\text{def}}{=} \frac{\rho(t)(r)}{\delta(t)}$$

Example. Considering our running example (Figure 4.13), three tasks are making use of resource admin: tasks Fill in admin. doc., Check payment reception, and Log information. According to their usage of this resource and their duration, their score is respectively $\frac{1}{30} = 0.0\overline{33}$ for task Fill in admin. doc., $\frac{1}{5} = 0.2$ for task Check payment reception, and

$\frac{1}{25} = 0.04$ for task Log information. Thus, task Check payment reception will be the first task put in sequence, followed by task Log information and finally task Fill in admin. doc.

4.5.4 Sequencing of Non-Parallelisable Tasks

This step applies whenever some tasks of the process have been tagged for sequencing. Sequencing a task consists in trying to remove it from the parallel gateway(s) to which it belongs, if any. In Section 4.3, we saw that, in order to preserve the trace semantics of the original process, a task could be moved only under its boundary. Consequently, the sequencing of a task must happen under its boundary. This means that, if, for instance, the boundary of the task belongs to a parallel gateway, this task will remain in parallel. Unfortunately, this is the price to pay in order to preserve the structural semantics of the process described in Section 4.3. Then, the task is simply put in sequence of any node of its boundary using the first pattern provided in Definition 4.14. Among the generated sequence graphs, the one offering the best trade-off between stochastic worst-case execution time and absorbance of the underrepresented resource is kept as best candidate. The operation is repeated for all the tasks that must be sequenced. Finally, the sequence graph obtained after the last task sequencing is transformed into its corresponding BPMN process, and returned to the user.

Example. Given our running example (Figure 4.13) and the results of the absorbance computation, we saw that one or several tasks making use of resource admin have to be sequenced. Moreover, we know the order in which they should be considered: task Check payment reception is the first candidate, task Log information the second, and task Fill in admin. doc. the third. Considering the optimal version of our sequence graph, we can see that task Check payment reception is already in sequence at the topmost level of its boundary: the body of the loop the structure. Thus, we can not make it more sequential. Task Log information however, is in parallel of all the remaining tasks. It can consequently be put in a new sequence node, inserted before, after, or between the two sequence nodes composing the subgraph of the main graph. By moving task Log information this way, each possible case will obtain the same stochastic worst-case execution time. Thus, the decision criterion will be the absorbance of the process. Unfortunately, although reducing the spike of requirements for resource admin to 3 parallel demands, the generated sequence graphs all kept an absorbance of ∞, because the two replicas of this resource remain requested 100% of the time. Thus, the last task making usage of this resource, namely, task Fill in admin. doc., has to be sequenced. Similarly to task Log information, this task does not belong to a conditional structure and can thus be placed at the top level of the main sequence graph. However, to preserve its dependency with, for instance, task Deliver by drone, it has to remain before the choice structure containing that task. Thus, it can be put before or between the two sequence nodes composing the subgraph of the main sequence graph. As for task Log information, the two possible graphs have the same stochastic worst-case execution time. Their absorbance will then be used to pick the best one. The

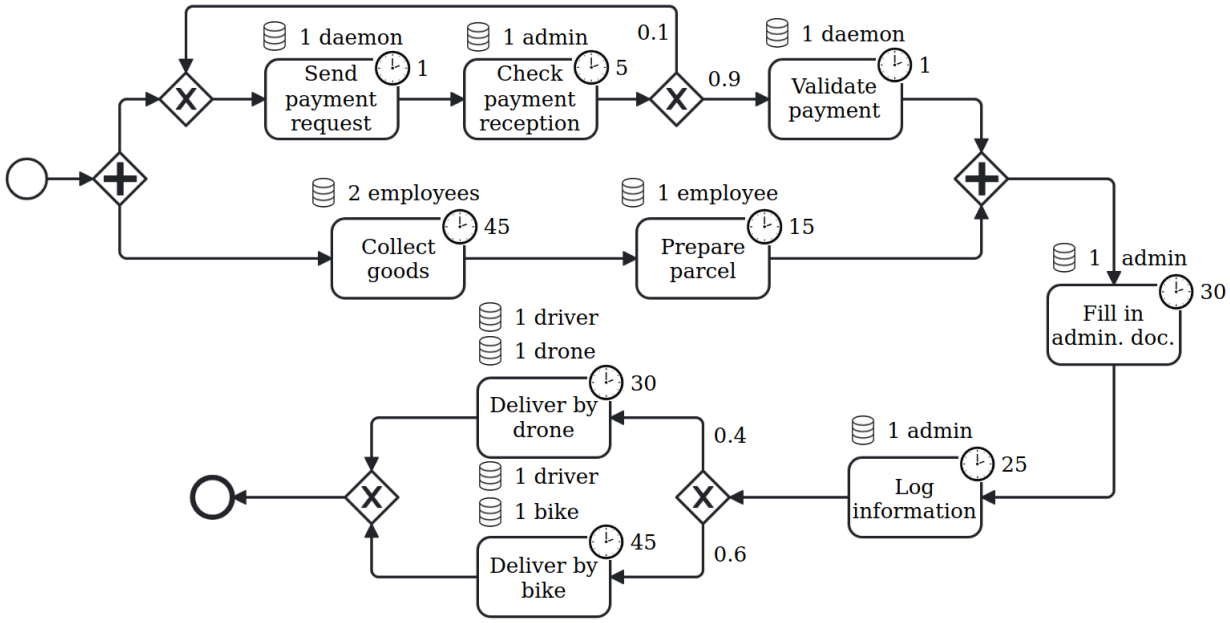


Figure 4.18: Refactored Process Returned by the Approach

one with the smallest absorbance[1] is the one in which both tasks Fill in admin. doc. and Log information are put in sequence between the two nodes composing the main sequence graph. This sequence graph is then converted to its equivalent BPMN process, represented in Figure 4.18. Compared to the original process, the generated one obtains an AET of 571m (9h31m), which shows an optimisation of 25.5%.

4.5.5 Pros and Cons

This approach has several advantages and drawbacks that are discussed in this section.

Pros

The main advantages of this approach are:

— its efficiency, which is ensured by performing only static analyses of the processes, thus avoiding the overhead caused by simulation;
— its computations and results regarding the pool of resources required by the process. This allows not only to change the position of the tasks in the process, but also to adjust—and in particular, reduce—the pool of resources so as to make it compliant with the needs of the process.

[1] Actually, the one with the smallest numerator, as the denominator of the fraction is still 0.

Cons

The main drawbacks of this approach are:

— its "one-shot" monolithic way of application. Indeed, the final BPMN process may end up completely restructured, potentially decreasing the capacity of the user to understand all the performed changes;
— its restriction to constant durations for tasks and IAT. In practice, it often happens that a task does not have a constant duration, but a duration that may vary slightly depending on its hidden parameters which make it inherently more or less complex from time to time;
— its imprecision in case of resources overusage. As no simulation is involved, the static analysis reaches some limitations when some tasks have to be sequenced. Indeed, there is no guarantee regarding the fact that the process having the shortest execution time, or the smallest absorbance, is the one obtaining the shortest AET.

4.6 Non-fixed Durations Step-by-Step Approach

The approach presented in this section differs from the previous one on two main aspects. Unlike the previous approach, the one presented in this section does not apply the refactoring operations once and in a monolithic way, but step by step, each of them being proposed and validated by the user. Consequently (s)he is more likely to understand the changes made to the process, and thus the resulting one. The second difference concerns the durations contained in the process, and more precisely the way in which they are represented. Indeed, we saw in the previous approach that only durations following a constant distribution were supported, which had an impact on the tasks of the process, and on its IAT. Moreover, this assumption was mandatory for the approach to work, as it allowed us to know statically which tasks were executed at any given time of the execution of the process. In this approach, we expand the support of the durations to any value following a probabilistic distribution.

These two changes implied to rethink entirely the approach. Indeed, the fact that durations are not constant anymore forces us to put aside any static analysis of the process, as these durations are only known at execution time. Moreover, the integration of the user in the loop at the key steps of the refactoring method implies a step-by-step modification of the process, where the number of changes between two processes coming from two successive steps must be minimal. This new version of the refactoring approach thus works the following way: each task of the original process is moved one by one, so that the deviation between two processes coming from two successive steps is minimal. The tasks of the process are moved only once, thus ensuring the termination of the refactoring approach. The user is solicited at two crucial steps of the approach: the choice of the task to move, and the choice of the process to keep (the original one or the modified one). In these two cases, the user can either validate the proposal, or decline it. If (s)he declines a task,

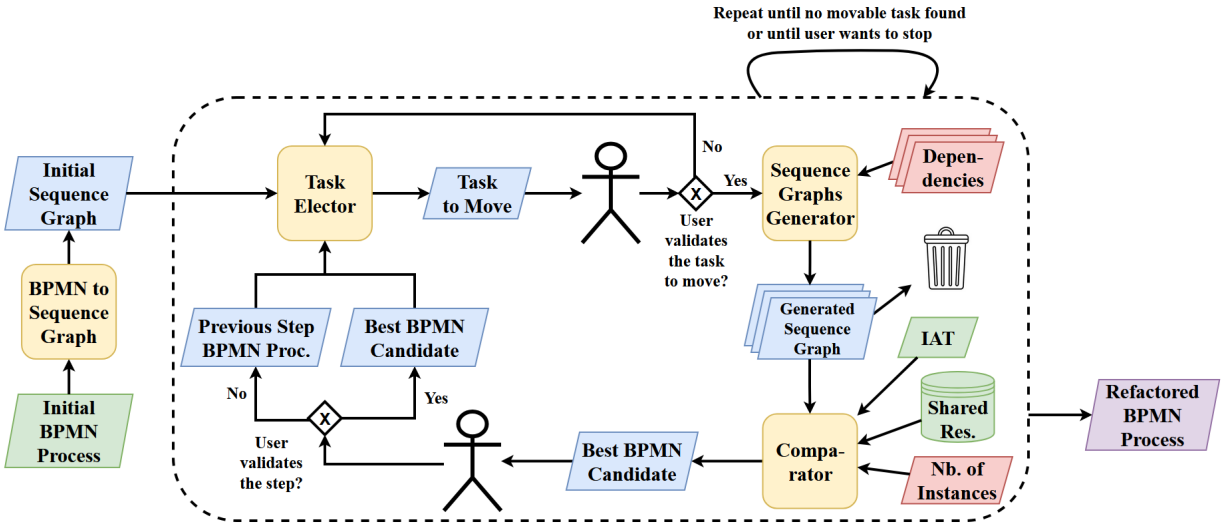


Figure 4.19: Overview of the Approach

another task is proposed to her/him, provided that not all the tasks of the process have already been moved. If (s)he declines a process, it is replaced by the previous process (i.e., the one obtained in the previous step, or the original one if not steps were performed yet). These steps are recalled in Figure 4.19. It is worth reminding that this approach is entirely based on the refactoring patterns presented in Section 4.2. Thus, this section only introduces the concepts specific to the given approach, without detailing the already presented notions.

Running example. The running example used throughout this section is a version of the BPMN process shown in Figure 4.1 where tasks have been given possibly non-constant durations, unlike in the previous approach. It is given is Figure 4.20. For instance, the duration of task Collect goods follows a normal distribution of parameters $\mathcal{N}(45, 5)$, meaning that it will range approximately between 23m and 67m, with an average duration of 45m. Similarly, the inter-arrival time of the instances of the process can be non-constant, and consequently follows a normal distribution of parameters $\mathcal{N}(25, 1.5)$. As durations and IAT follow either constant, normal, or uniform distributions, the AET over 100 instances of the process may vary significantly from one simulation to another. For this reason, the 100 instances of the process are simulated 100 times. The metrics of these 100 simulations are then averaged, and used as basis for computation. Under these assumptions, the AET of the original process is 750m (12h30m). A keen eye will notice that this duration is much larger than the sum of the worst durations of the tasks of the process. This is due to the fact that resource admin only has 2 available replicas for all the running instances, and that several tasks are making use of it. Consequently, the instances will often have to wait for a replica of resource admin to be released. This waiting time will, in the end, increase the execution time of the instance, and thus the AET of the process.

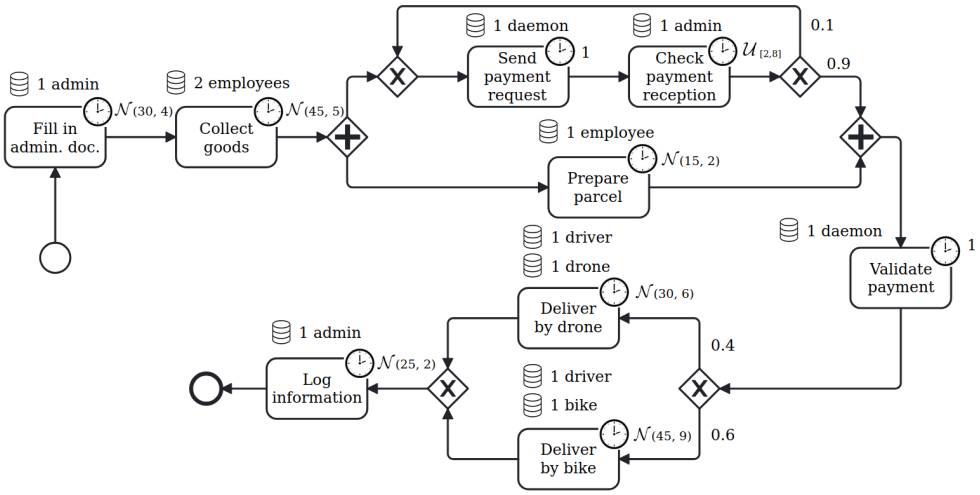


Figure 4.20: Running Example with Non-Constant Durations

4.6.1 Task Election

The first step of the iterated part of this approach consists in electing a task of the process as candidate for the next move. This selection can be done in several ways: randomly, greatest average durations first, greatest average durations last, greatest resources usage first, greatest resources usage last, a mix of several criteria, etc. Based on our experiments, we observed that the selection method giving the best results in terms of optimisation in this context is the one selecting tasks by ascending order of average duration. The intuition behind the good quality of this selection method resides in the fact that the duration of a task is directly related to the targeted optimisation criteria which is the AET of the process, and in the fact that the tasks of the process are moved once and only once. Consequently, finding the best position of a task is crucial in order to reach the greatest optimisation. However, the earlier a task is moved, the more its position will be subject to changes, due to the shift of the remaining tasks. As tasks with longer durations have a greater impact on the execution of a process than tasks with smaller durations, their position, once found, has to remain as stable as possible. Thus, tasks with longer durations should be moved later than tasks with shorter durations. Once a task has been selected, it is proposed to the user. If the user accepts to move this task, the refactoring patterns presented in Section 4.3 are applied to the task and its boundary in order to generate all the possible abstract graphs corresponding to a different position of this task.

Example. Given the running example shown in Figure 4.20, the tasks Validate payment and Send payment request will be moved first, followed by task Check payment reception, and so on.

4.6.2 Computation of the Best Refactoring Steps

Computing the best refactoring step given a task to move can be done in many different ways. A naive—yet natural—way of computing it would be to generate all the possible processes corresponding to a different position of a given task using the refactoring patterns presented in Section 4.3, and then repeating the operation on all the generated processes, until all the tasks of the process have been moved. This option would generate an arborescence of processes, whose leafs would represent all the possible dispositions of the tasks in the process. Then, by comparing these leafs, we would be able to find the process obtaining the shortest AET. Finally, by traversing the arborescence backward, up to the original process, we would obtain the list of successive steps to apply to our process in order to reach the optimal one. However, such an exhaustive exploration, despite returning the optimal process, is not applicable in a real-time context. Concretely, for a process containing 15 tasks and in which each task can have 20 different valid positions, the corresponding arborescence would contain $15^{20} = 3 \times 10^{23}$ leafs. As simulating a single process takes at least milliseconds, simulating such an important number of processes is not feasible in a reasonable time. Consequently, heuristics are required to reduce the size of the arborescence, that is, its number of nodes.

Heuristics.

The idea of a heuristic is to explore efficiently the state-space, that is, the space of all possible solutions. In our approach, not only the exploration of the arborescence is costly, but also its construction, as it may end up containing billions of billions of nodes. Thus, both the exploration and the construction of the arborescence must be bounded. Several methods can be used to bound the exploration and the construction. A first option could be to bound the construction of the arborescence to a given number of steps, after which the process with shortest AET would be kept. Another option could be to keep several best candidates at each step, based on a score attributed to each candidate. A third option could be to try to make a structural analysis of the process, evaluating the balance of parallelism and sequence in it, or its potential to be a good solution. Among the presented ones and based on our experiments, the option giving the best trade-off between quality of the result and execution time is the one in which several best processes are kept at each step, their quality being based on a score attributed to each of them. This score is based on the metrics obtained by simulating each generated process of the given step. Then, a number n of processes obtaining the highest scores are kept and used as basis for the next step.

The computation of the score is based on variations of two different metrics: the AET and the resources usage of the process. It also makes usage of the metrics of the ancestor processes of the current one to adjust the score. The AET is declined in three different metrics: the *AET mean difference*, the *AET standard deviation difference*, and the *AET local difference*.

Definition 4.28 (AET Mean Difference)**.** *Let* $G_{n+1} = (V_{n+1}, E_{n+1}, \Sigma_{n+1})$ *be a BPMN process generated at step* $n+1$*, and let* $S_G = \{G_1, ..., G_n\}$ *be the sequence of BPMN processes generated in the* n *previous steps. The* AET mean difference *of* G_{n+1} *is defined as*

$$\delta_{\mu_{AET}}(G_{n+1}) = \mu_{AET}(G_1, ..., G_n) - \mu_{AET}(G_1, ..., G_{n+1})$$

where

$$\mu_{AET}(G_1, ..., G_k) = \frac{1}{k} \sum_{i=1}^{k} AET(G_i)$$

Definition 4.29 (AET Standard Deviation Difference)**.** *Let* $G_{n+1} = (V_{n+1}, E_{n+1}, \Sigma_{n+1})$ *be a BPMN process generated at step* $n+1$*, let* $S_G = \{G_1, ..., G_n\}$ *be the sequence of BPMN processes generated in the* n *previous steps. The* AET standard deviation difference *of* G_{n+1} *is defined as*

$$\delta_{\sigma_{AET}}(G_{n+1}) = \sigma_{AET}(G_1, ..., G_n) - \sigma_{AET}(G_1, ..., G_{n+1})$$

where

$$\sigma_{AET}(G_1, ..., G_k) = \sqrt{\frac{\sum_{i=1}^{k} (AET(G_i) - \mu_{AET}(G_1, ..., G_k))^2}{k}}$$

Definition 4.30 (AET Local Difference)**.** *Let* $G_n = (V_n, E_n, \Sigma_n)$ *be a BPMN process generated at step* n *and let* $G_{n+1} = (V_{n+1}, E_{n+1}, \Sigma_{n+1})$ *be a BPMN process generated from* G_n*. The* AET local difference *of* G_{n+1} *is defined as*

$$\delta_{AET}(G_{n+1}) = AET(G_n) - AET(G_{n+1})$$

The resources usage of the process is declined in two different metrics, namely, the *resource usage mean difference* and the *resource usage local difference.*

Definition 4.31 (Resources Usage Mean Difference)**.** *Let* $G_{n+1} = (V_{n+1}, E_{n+1}, \Sigma_{n+1})$ *be a BPMN process generated at step* $n+1$*, let* $S_G = \{G_1, ..., G_n\}$ *be the sequence of BPMN processes generated in the* n *previous steps, and let* P *be the available pool of resources of* G_{n+1}*. The* resources usage mean difference *of* G_{n+1} *is defined as*

$$\delta_{\mu_{res}}(G_{n+1}) = \mu_{res}(G_1, ..., G_n) - \mu_{res}(G_1, ..., G_{n+1})$$

where

$$\mu_{res}(G_1, ..., G_k) = \frac{1}{k} \cdot \frac{1}{|P|} \sum_{i=1}^{k} \sum_{r \in P} \rho(G_i)(r)$$

Definition 4.32 (Resources Usage Local Difference)**.** *Let* $G_n = (V_n, E_n, \Sigma_n)$ *be a BPMN process generated at step* n*, let* $G_{n+1} = (V_{n+1}, E_{n+1}, \Sigma_{n+1})$ *be a BPMN process generated from* G_n*, and let* P *be the available pool of resources of* G_{n+1} *The* resources usage local

difference *of* G_{n+1} *is defined as*

$$\delta_{res}(G_{n+1}) = \frac{1}{|P|} \sum_{r \in P} \rho(G_{n+1})(r) - \rho(G_n)(r)$$

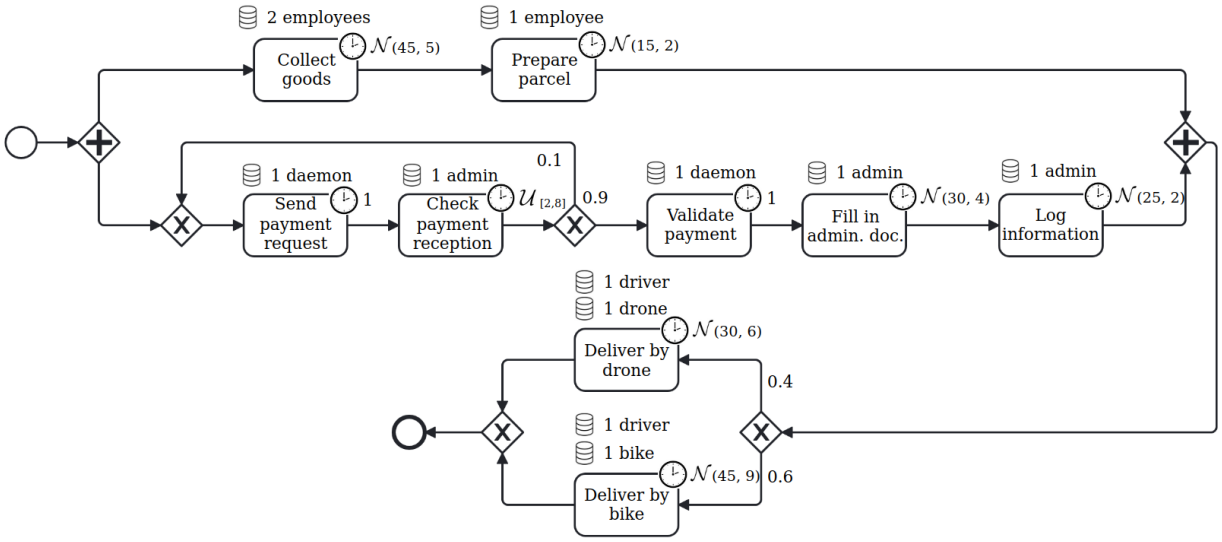
These five metrics are then normalised in order to obtain a value between 0 and 1 for each of them. Finally, the *score* of the current process is computed as the weighted sum of these five metrics, proportional to the resources usage and inversely proportional to the AET.

Definition 4.33 (Score of a Process)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process, let* $\delta_{\mu_{AET}}$, $\delta_{\sigma_{AET}}$, δ_{AET}, $\delta_{\mu_{res}}$ *and* δ_{res} *be the five previously defined metrics computed on* G*, and let* ω_{AET}, ω_{res}, $\omega_{loc} \in \mathbb{R}^+$ *be the weights respectively attributed to the metrics based on AET, resources usage and local information. The score of* G *is defined as*

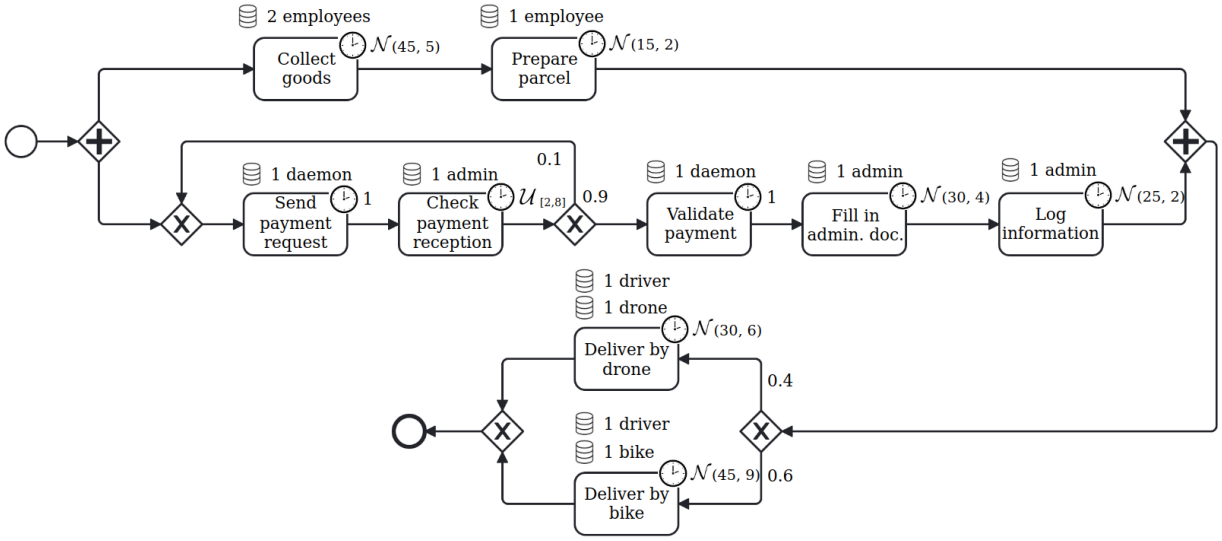
$$score(G) = \omega_{AET} \times (\delta_{\mu_{AET}}(G) + \delta_{\sigma_{AET}}(G) + \omega_{loc} \times \delta_{AET}(G)) + \omega_{res} \times (\delta_{\mu_{res}}(G) + \omega_{loc} \times \delta_{res}(G))$$

The weights attributed to the metrics can be given by the user as an input of the approach. If none are given, the default ones, based on our experiments, are $\omega_{AET} = 0.6$, $\omega_{loc} = 0.5$, $\omega_{res} = 1.0$. Once each graph of the current step is attributed a score, the n best ones, where n is a parameter of the approach, are kept as best candidates for the next step. The possible positions of a new task are then computed on these n processes, and so on, until all the tasks of the process have been moved. Finally, the process obtaining the shortest AET is elected as best process, and the sequence of intermediate steps necessary to obtain it is computed backward, up to the current step. The immediate next step (i.e., new process) is then proposed to the user, who can either accept or decline it.

Example. Figure 4.21 presents two refactored versions of the running example shown in Figure 4.20. The first one (Figure 4.21(a)) is the process obtained by applying the aforementioned heuristic, while the second one (Figure 4.21(b)) is the optimal version of the running example, obtained with a full generation and exploration of the arborescence. The refactored process obtained with the heuristic has an AET of 502m (8h22m), which represents a 33.1% of optimisation with regards to the original version. Interestingly, it does not contain that much parallelism in it. Moreover, all the tasks requiring the same resource were not put in parallel, e.g., tasks Fill in admin. doc. and Log information. For these two tasks, it is likely to result from the stress of resource admin, which has only two available replicas. However, for resource employee, this may be a trade-off between resource usage and uselessness of adding parallelism, as the parallel branch containing tasks Collect goods and Prepare parcel in sequence usually executes faster than the other parallel branch. This example is a very good illustration of the quality of the heuristic, as, as the reader can see in Figure 4.21(b), the refactored process returned by the full exploration (which took several days of computation and generated approximately 300k processes) is strictly identical to the one returned by the heuristic.



(a) Refactored Version of the Running Example Obtained with Heuristics



(b) Refactored Version of the Running Example Obtained with Full Exploration

Figure 4.21: Refactored Versions of the Running Example

4.6.3 Pros and Cons

This approach has several advantages and drawbacks that are discussed in this section.

Pros

The main advantages of this approach are:

— its step-by-step user-validated progression, which makes the user more likely to understand the changes happening to the process;
— its management of non-fixed durations, which makes it one step closer to reality as it is able to model more realistic behaviours;
— its precision in the generated results. Indeed, thanks to simulation, the results computed by this approach are very precise, which plays an important role in the quality of the generated process;
— its efficiency, as the heuristics allow one to optimise processes powerfully in a very short time.

Cons

The main drawbacks of this approach are:

— its involvement of the user, which may drastically lower the quality of the refactored process, as (s)he may decline some tasks, or some key processes that do not seem relevant for her/him;
— its lack of information regarding the optimal pool of resources required by the process to run without latencies;
— its usage of simulation, which, in some not-so-edgy cases, may take a significant amount of time to simulate a process sufficiently enough to provide reliable metrics.

4.7 Multi-Objective Approach

The last proposal that we made on this topic aimed at enlarging the scope of the refactoring approach to multiple optimisation objectives. Indeed, it is frequent for a company to want to optimise not only the execution time of its business processes, but also their resources usages, or their costs. Moreover, these metrics are easily computable using the simulation techniques already used in the previous section. It is worth reminding that this approach is entirely based on the refactoring patterns presented in Section 4.2. Thus, this section mostly presents the differences induced by the multiple objectives with regards to the single objective case, without detailing the already presented notions.

4.7.1 Multi-Objective Optimisation Problem

Generic Idea

A multi-objective optimisation problem, in opposition to a single objective one, is an optimisation problem in which several parameters should be optimised. In this context, one problem may admit several optimal solutions, each of them providing a better improvement for some parameters, and a worse one for the others. Such solutions are called *Pareto optimal solutions*. The quality of a solution is based on a criteria called *Pareto dominance*, which states that a solution s *Pareto dominates* a solution s' whenever all the parameters of s are greater or equal to the parameters of s', except at least one parameter that is strictly greater in s than in s'.

Application to our Problem

In this context, an optimisation criterion is any criterion that can be computed using simulation. In this thesis, we will focus on the AET of a process, its resources usage, and its cost. Moreover, we slightly deviate from the original definition of a multi-objective problem as it can be found in the literature, as we allow the weighting of the optimisation criteria. This permits, in our opinion, some more realistic behaviours, where optimising the AET of the process can be more important than optimising the usage of one given resource. However, this requires an adjustment of the notion of Pareto dominance. Indeed, in this context, a process may dominate another one even though some of its optimisation criteria are lower than the ones of another process. For this reason, we define a notion of *dominance score* that aggregates the values of the multiple optimisation criteria of the process into a single value, which takes care of the prevalence of some criteria with regards to some others, thus allowing a proper comparison.

Definition 4.34 (Dominance Score)**.** *Let G_0 be the original process, let G_1, G_2 be two generated processes, and let C be a set of optimisation criteria. G_1 is said to dominate G_2 if and only if*

$$\sum_{c \in C} \omega_c \times \frac{c(G_0)}{c(G_1)} > \sum_{c \in C} \omega_c \times \frac{c(G_0)}{c(G_2)}$$

where ω_c is the weight of the optimisation criterion c, and $c(G_i)$ is an operator returning the value of the optimisation criterion c on the graph G_i, $i \in \{0, 1, 2\}$.

Example. The running example used in this section is the same than the previous section's one, that is, the one shown in Figure 4.20. The IAT also follows the same normal distribution of parameters $\mathcal{N}(25, 1.5)$, and 100 instances of the process are being run. However, here, several criteria of optimisation are driving the refactoring process, namely, its AET, its cost, and its usage of resources admin, employee, and driver. Moreover, these criteria are weighted, giving each a variable importance in the optimisation. The AET gets a weight $\omega_{AET} = 30\%$, the cost a weight $\omega_{\$} = 20\%$, the usage of resource admin a weight $\omega_a = 20\%$, the usage of resource employee a weight $\omega_e = 20\%$, and the usage of resource driver a weight

$\omega_d = 10\%$.

4.7.2 Task Election

The idea of task election presented in the previous section is not applicable anymore. Indeed, this idea arose from two hypothesis: (i) classifying the tasks of the process could be done based on a quantitative metric, and (ii) there was a single optimal version of the process. Here, several optimisation criteria are considered. Thus, the optimisation can be performed in several directions: focus on a single criteria, several of them, a balance between several, etc. Consequently, finding a statically computable metric usable for classifying and comparing the tasks of the process becomes harder. Moreover, the overall impact of a given task becomes more unclear, as a task may have a strong impact on some criteria and a small one or none on some others. Regarding the second point, in such context, we no longer have a single optimal version of the process, but several ones, called Pareto optima. All these optima have different values for their optimisation criteria, yet the same overall optimisation. These optimal versions are said to be on the *Pareto front*, a curve providing the set of parameters values giving a maximal optimisation. For these two reasons, comparing and assessing the impact of a task with regards to its siblings becomes challenging, and does not show much interest in this approach. Thus, the task election mechanism employed in this approach randomly picks the task to move. As far as our experiments led us, this simplification does not impact the approach, as more complex election mechanisms were obtaining the same results than the random one.

4.7.3 Generation of the Best Refactoring Steps

The generation of the best refactoring steps remained quite similar to the single objective one. Indeed, when a task is elected, the refactoring patterns are applied on it so as to generate all the possible positions of this task in the process. Then, the n processes obtaining the best optimisation scores are kept as best candidates, and the refactoring patterns are applied again on them, with a new task to move. However, similarly to the task election phase, it became more complex to find good heuristics for traversing the arborescence of solution efficiently, or to prune it. Thus, it became trickier to navigate efficiently through this arborescence, and consequently to keep the best (or at least good) candidates at each step. For this reason, we decided to discard the step-by-step approach, which allowed us to remove the bound on the number of steps originally required for termination. This gain of freedom allowed us to make use of several well-known algorithms that we applied to our approach.

4.7.4 Optimisation Algorithms

Multi-objective evolutionary algorithms (MOEAs) are specifically designed to address such optimisation problems by efficiently exploring the state space and maintaining a diverse

set of non-dominated solutions. Throughout the years, well-known single objective optimisation algorithms which had proven themselves, such as *hill climbing*, or *simulated annealing* [KGV83], were slightly modified to support multi-objective problems. As a basis of analysis, we make us of these two algorithms and apply them to our refactoring problem. The hill climbing algorithm iteratively applies refactoring operations to the current process, and keeps the best version between the original and the refactored one. If the two processes provide the same optimisation, the new one is kept in 50% of the cases. The simulated annealing algorithm performs almost identically, except that it may keep a solution that provides a lower optimisation than the current one (i.e., a worse solution), so as to promote diversity. The probability of keeping such a solution decreases over time according to a cooling schedule.

Another class of algorithms, named *evolutionary algorithms* (EAs) [Bäc96], have proven themselves as particularly effective in the field of multi-objective optimisation. Evolutionary algorithms are population-based metaheuristics inspired by natural evolution, using mechanisms such as mutation, selection, and sometimes crossover to make a set of candidate solutions evolve over time. Well-known families of multi-objective evolutionary algorithms include NSGA-II [DPAM02], SPEA2 [ZLT01], SMS-EMOA [BNE07], AGE-MOEA [Pan19], AGE-MOEAII [Pan22], ESPEA [BSS15], PAES [KC99], and others.

In this work, we leverage two MOEAs—NSGA-II and PAES—to explore the space of refactored processes. Each individual in the population directly encodes a process model, as in [Str17, BZS18], rather than a vector of parameters. A mutation corresponds to applying a refactoring pattern (i.e., moving a task), with a 75% probability per individual. If the mutated individual is worse than the original one, it is retained with a small probability (5%) to promote diversity and avoid premature convergence. The population is initialised with random mutations of the original process, and the algorithms maintain an archive (or population) of non-dominated solutions throughout the search. The stopping criterion is based on a fixed execution time. Comparisons between individuals use a non-dominance criterion, aggregating objectives into a score to determine dominance.

NSGA-II [DPAM02] is one of the most widely used and effective multi-objective evolutionary algorithms. Building on the general evolutionary principles described above, NSGA-II distinguishes itself through its fast non-dominated sorting and crowding distance mechanisms, which help maintaining diversity and guiding the population toward the Pareto front. While NSGA-II typically leverages both mutation and crossover to explore and exploit the search space, only mutation is applied in our context, due to the challenges of defining meaningful crossover for process models. This adaptation may affect the algorithm's ability to fully utilise its strengths, but NSGA-II remains a strong baseline for multi-objective optimisation in complex domains. In contrast, PAES [KC99] is a simpler evolutionary algorithm that relies solely on mutation and maintains an archive of non-dominated solutions to guide the search. The archive is updated whenever a new non-dominated solution is found, and Pareto dominance is used to compare solutions. This approach avoids premature convergence and helps approximating the Pareto front by en-

suring diversity among solutions, making it particularly effective when crossover is not available or not meaningful.

Finally, as a comparison basis, the fast algorithm presented in Section 4.6 is also used. It slightly differs from its original version, as it now has to support multiple optimisation objectives. To do so, it aggregates the metrics of the considered process in the form of a score that is used for comparison. We recall that this algorithm moves the tasks of the process only once, and keeps only the best solution at each refactoring step.

Example. As an example, let us show the BPMN process obtained by applying the NSGA-II algorithm to the running example. The process resulting from the refactorisation carried out by this algorithm is shown in Figure 4.22. This process, reaching 31.7% of optimisation, exhibits several interesting properties, similar to the ones presented in Section 4.6.

First, we can see that none of the tasks necessitating resource admin were put in parallel. This can be explained by the important usage of this resource all over the lifetime of the process, which is itself explained by the small number of available replicas of this resource (2). Thus, it is likely that putting these tasks in parallel would have generated important synchronisation delays compared to their sequential version.

The second interesting thing that can be noticed is that the tasks requiring the resource employee never appear in parallel neither, even though 6 replicas of this resource are available. There might be several explanations for this to appear. A possibility is that, on average, the other branch of the parallel gateway executing the tasks Collect goods and Prepare parcel does not execute in less time than the other. Thus, putting the tasks Collect goods and Prepare parcel in parallel would not lower the execution time of the process, although increasing its pike usage of resource employee. Another possibility is that, due to the IAT of the process in our example, more than three instances of it may be executing at the same time, and more precisely, may be executing tasks Collect goods and/or Prepare parcel at the same time. Consequently, the process would require more than 6 replicas of resource employee at the same time. This would delay the execution of the tasks Collect goods and Prepare parcel, and thus increase the AET of the process. The simplest solution is thus to keep those two tasks in sequence.

4.7.5 Pros and Cons

This approach has several advantages and drawbacks that are discussed in this section.

Pros

The main advantages of this approach are:

— its handling of multiple objectives, which makes it one step closer to reality;
— its multiple types of bounds for the optimisation algorithms (number of iterations, number of tasks, time, ...);

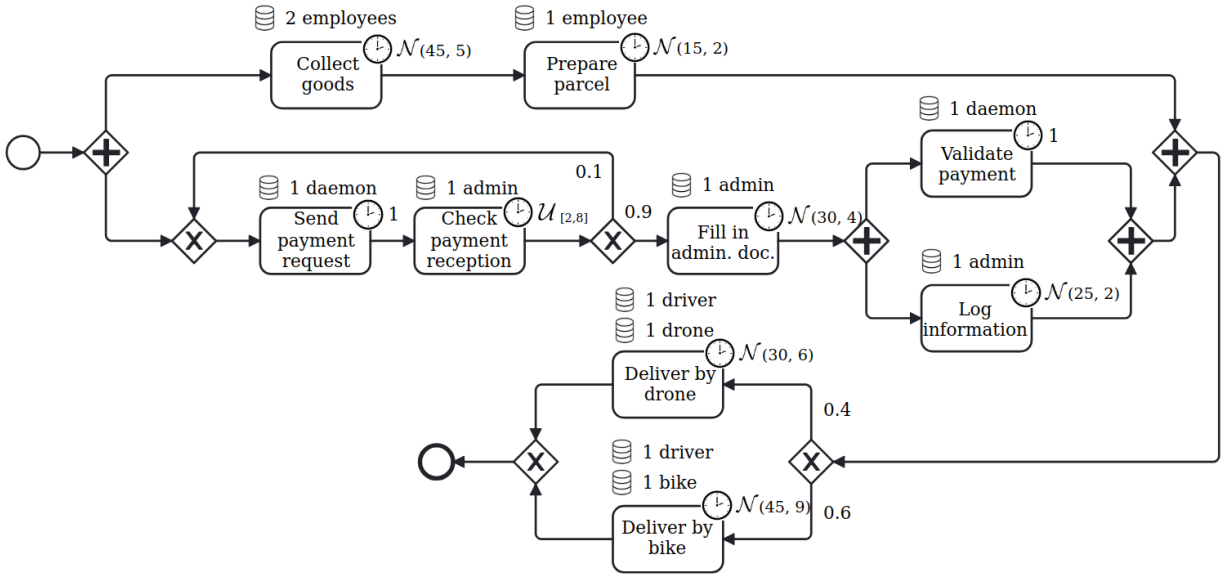


Figure 4.22: Refactored Goods Delivery Process

— it handling of non-fixed durations.

Cons

The main drawbacks of this approach are:

— its impossibility to rely on well-suited heuristics to speed up the computations anymore;
— its lack of, to the best of our knowledge and experiments, perfectly suitable optimisation algorithm for navigating through the space of possible solutions, leading to non-optimal results;
— its usage of simulation.

4.8 Conclusion

In this chapter, we have presented three different approaches aiming at tackling the problem of optimising business processes. These three approaches have a common foundation, which is the employed technique: process refactoring. All of them rely on refactoring patterns, useful for providing strong semantics preservation guarantees. These approaches all have their share of advantages and drawbacks, which are discussed at the end of their respective sections. As a witness to their quality, and for the sake of experimentation, they were all implemented and tested. The prototypes and the conducted experiments are detailed respectively in Sections 5.3.1, 5.3.2, and 5.3.3 of this manuscript.

Chapter 5

Tools & Experiments

"It doesn't matter how beautiful your theory is, it doesn't matter how smart you are. If it doesn't agree with experiment, it's wrong."

Richard Philips Feynman

Contents

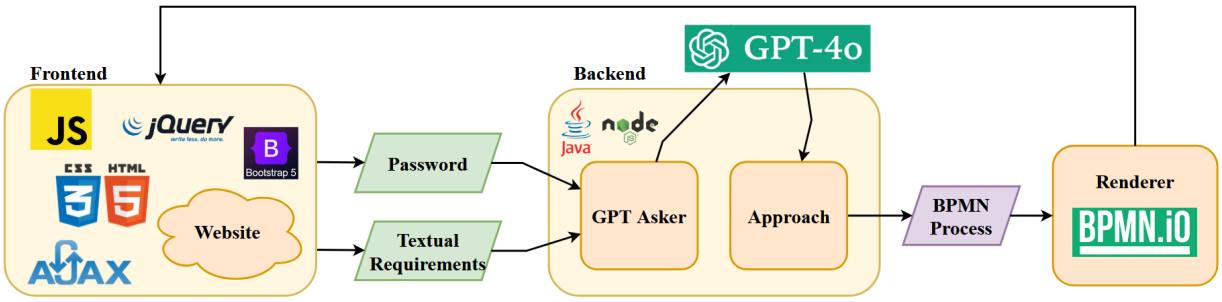


Figure 5.1: Overview of the Toolchain

This chapter presents the various tools that were developed to perform experimental studies for each approach presented in Chapters 3 and 4. It also presents the GIVUP tool [NS25], which extends the BPMN modelling approach presented in Chapter 3 with verification techniques. These tools represent in total approximately 45k lines of Java code. All the experiments presented in this section were conducted on a HP EliteBook x360 1030 G8 Notebook PC running with an Intel Core i5-1145G7 @ 2.60GHz VPRO and 16GB of RAM, unless otherwise stated.

5.1 Modelling of BPMN Processes

5.1.1 Tool

The approach presented in Chapter 3 has been entirely implemented as a tool written in Java and consisting of approximately 12k lines of code. To facilitate its usage, the Java code has been embedded in the backend of a web server which is freely available online, along with the different datasets used.[1] The implementation details are given in Figure 5.1. The user writes her/his textual description on the web application that is developed in HTML, CSS, JavaScript and makes use of JQuery, Ajax and BootStrap. The description is then transmitted to the backend written in NodeJS, which asks the Java program to send the description to GPT. The expressions returned by GPT are transformed into a graph, which is eventually converted into the resulting BPMN process following the multiple steps detailed in Chapter 3. This process is finally rendered by bpmn.io[2], and displayed in the web application.

5.1.2 Evaluation

The evaluation presented in this section is separated in three different parts. The first one consists in comparing the tool proposed in this approach to other tools coming from the literature, and to LLMs prompted directly. The second one provides an analysis of the

[1] https://github.com/QuentinNivon/Text_to_BPMN
[2] https://bpmn.io/

Component	Prompt Content
Agent Role	`You are an expert in Business Process Model and Notation (BPMN), and your role is to transform the textual description that will be given to you as input between curvy brackets into the corresponding BPMN process.`
Expected Output Format	`For readability, the format that you should adopt is a visual representation, that you can display directly in your answer. You should restrict yourself to the following elements of the BPMN syntax: tasks or activities, sequence flows, parallel gateways, and exclusive gateways.`
Advice of Undesired Behaviour	`This means that your answer should not contain, for instance, inclusive gateways.`

Example of Correct Output	`Given the input {The goods must be packaged before being sent}, an example of correct input could be:`
	`---------- ----------`
	`| PackageGoods | --> | SendGoods |`
	`---------- ----------`

Figure 5.2: Prompt Used for LLM Direct Usage

incorrect results obtained by this approach in order to understand the reasons leading to the failure. The third one gives insights on the behaviour of this approach when the tasks of the description given as input are not already named.

Tools Comparison

To the best of our knowledge, there are only two tools aiming at generating BPMN processes from natural language descriptions available online at the moment: ProMoAI [KBSvdA24b] and NaLa2BPMN [EAA+24]. Our tool was compared to them, and also to Gemini [ea24a] and GPT-5 [ea24b] prompted with a rigorous prompt following the best practices in terms of prompt engineering [Bro20, MIT23, MIST24, XYL+25], which is given in Figure 5.2. The focus was made both on the accuracy of the results and on the time taken by each tested tool to generate the BPMN process. 200 descriptions were used, coming from various sources. 25% come from the literature (PET dataset [BvdAD+22] and proceedings). The remaining 75% were handcrafted by 9 users (5 experts having between 3 and 15 years of experience with BPMN and 4 novices not used to BPMN) who experimented the tool. All these examples contain tasks which were named beforehand.

The results of these experiments, presented in Table 5.1, are split into three different groups. The first group, labelled with a tick, represents the processes that were considered as valid by the two experts who analysed the results, called reviewing experts. Here, the notion of validity relies on the correspondence between the expected process and the generated one. A process is considered valid if it corresponds exactly to the expectations of the reviewing experts, and thus, to the textual requirements.

Table 5.1: Results of the BPMN Generation Experiments

Tool/Model	✓	?	✗	Avg. Ex. Time
Our tool	**83%**	9.8%	**7.2%**	**7.21s**
NaLa2BPMN	32.8%	8.9%	58.3%	68.7s
ProMoAI	50%	**8.7%**	41.2%	24.7s
Gemini	73.4%	13.8%	12.8%	7.67s
GPT-4-turbo	69.8%	19.3%	10.9%	11.8s

The second group, labelled with a question mark, represents the processes that were considered as ambiguous by the reviewing experts. Such processes are considered as ambiguous because, according to their textual description, one may generate several valid processes. As a choice has to be done among the multiple valid processes, one of them is generated, although it may not correspond to the expectations of the experts. For this reason, they belong to the group of ambiguous processes. For instance, a simple sentence such as "*I want A and B and C*" does not state how A, B, and C are related to each others. Thus, putting them in sequence, in parallel, or partially in sequence and in parallel remains correct with regards to the description. Similarly, a sentence such as "*I want A before B or C before D*" can be interpreted as a choice between "*A before B*" and "*C before D*", or as a sequence executing first A, then "*B or C*", and finally D. For this reason, such processes have been separated from the others, but remain considered as valid.

The third and last group, labelled with a cross, represents the processes that were considered as invalid by the reviewing experts. Such processes are at least partially non-compliant with the textual description. It is for instance the case when a non-ambiguous constraint is missing (e.g., two tasks are not put in sequence although they should be), or erroneous (e.g., two tasks are put in an exclusive choice instead of one after the other). Invalid processes are generated when GPT is not able to extract a constraint described textually, or when it misinterprets it.

The results of these experiments, provided in Table 5.1, showed that our tool obtains the best results both in terms of generation quality (with 83% of well-formed processes) and execution time (with an average execution time of 4.43s). Without much surprise, the execution time of this approach grows as the textual description grows, and as the number of generated expressions increases. Rather surprisingly, GPT-4-turbo and Gemini obtained very good results, especially with regards to their low failure percentage. However, the results obtained by the LLMs must be handled with care as they are very probably overrated. Indeed, to the best of our knowledge and experiments, LLMs are not yet capable of generating the XML code of a BPMN process correctly. For this reason, the LLMs were asked to generate a textual representation of the BPMN process, which was then visually analysed by the reviewing experts and used to compute the score of these models. As generating the exact XML code adds an additional difficulty layer to the LLM, it is likely that the results shown in Table 5.1 would be lower.

Incorrect Results Analysis

Table 5.1 shows that our approach does not manage to generate the expected process in 7.2% of the cases, which corresponds to 14 of our examples. The second part of these experiments thus consist in understanding the source of these mismatches, and quantify it. Such incorrect processes are caused by expressions returned by GPT not reflecting exactly the information contained in the description.

Table 5.2: Results of the Analysis of the 14 Incorrect Processes

	Missing constraint	Added constraint	Modified constraint
Total	**9**	**17**	**6**
Average	**0.64**	**1.21**	**0.43**

Table 5.2 summarises the results, given as a sum or an average. Column 1 shows the number of *missing constraints*. A missing constraint represents a condition described textually, and which does not appear in the generated expressions. For instance, if the description says that two tasks should appear in sequence, but no such constraint appears in the generated expressions, then there is one missing constraint. Column 2 gives the number of *additional constraints*. An additional constraint, by opposition to a missing one, represents a condition not described textually, but which appears in the generated expressions. For instance, if the description does not specify any constraint between two tasks, but they appear in sequence in the generated expressions, then there is one additional constraint. Column 3 presents the number of *modified constraints*. A modified constraint represents a relationship between two tasks that does not correspond to the one given in the description. For instance, if the description expresses that two tasks should be mutually exclusive, but they appear in sequence in the generated expressions, then there is one modified constraint. Row 1 describes the number of incorrect constraints obtained in total for each type, while row 2 represents the average number of incorrect constraints per process. Based on these three notions, we studied the expressions returned by GPT for these 14 incorrect examples. This table highlights a first interesting information which is that, even though being incorrect, these 14 processes are rather close to the expected ones as they only contain a few missing or incorrect constraints on average (approximately 2 incorrect constraints per incorrect processes). It is also interesting to note that the prevalence of the possible errors is not the same, as GPT has a tendency to add constraints between elements that are not constrained, while it is less likely to miss a constraint or misunderstand (i.e., modify) it.

Results with Unnamed Tasks

The results shown in Table 5.1 present experiments carried out on descriptions containing tasks that were named beforehand. To go further in the analysis of this approach, we decided to perform experiments on pure raw descriptions, in which the tasks were not

named beforehand. These experiments were conducted on 50 successfully generated examples among the 200 ones presented in Table 5.1 (i.e., they belong to Column 1). These examples were specifically chosen as they contain tasks which are not named meaninglessly (e.g., *A*, *BBBB*, or *Dummy*). The approach failed to generate 8 of them, which shows a degradation of 16% of the quality of the results when tasks are not named beforehand.

A deeper analysis of these examples allowed us to detect two main issues in the results returned by GPT. The first one is that GPT often misinterprets some parts of the text, thus leading to additional or missing tasks, which induce an incorrect process. The second issue resides in the capacity of GPT to map two distinct portions of text to the same task. Often, one introduces a task that has to be processed, and, later, references this task, as it has to be executed again. In such cases, GPT sometimes misses the fact that these two portions of text correspond to the same task, thus preventing the resulting process to contain, e.g., a loop. Finally, we observed that, the larger the description, the more GPT tends to make the aforementioned mistakes.

5.1.3 Threats to Validity

The experimental results presented in this section are inherently subject to threats. One threat that we identified is strongly connected to the training dataset used to fine-tune GPT. Indeed, this dataset was mostly handcrafted. This implies that, even though making many efforts to avoid it, the writing style of the used descriptions might be rather similar. As a consequence, it may be the case that descriptions written in a different style are less well understood by GPT, leading to less representative BPMN processes. This could potentially explain why GPT is sometimes performing better on some descriptions compared to some others. To mitigate this phenomenon, we try to gather descriptions coming from various sources, thus having different writing styles.

Another threat, somehow related to the previous one, is the size of the used datasets. Even though trying to make them rather large and varied, building and/or gathering textual descriptions and analysing them is tedious and time-consuming. Indeed, one must not only find/build valuable BPMN processes, but also their textual description and the set of expressions that should be produced by GPT. For these reasons, the training (resp. validation) dataset is for now limited to roughly 400 (resp. 200) examples. However, one of our main focuses is to keep growing these datasets.

5.2 Verification of BPMN Processes

This section presents an extension of the approach presented in Chapter 3, in which a process can be generated from a textual description, and also verified according to a property, also described textually. This contribution was thought as a tool, named GIVUP, which stands for "**G**enerat**I**on and **V**erification of **U**nderspecified **P**rocesses". GIVUP performs the model checking of the generated BPMN process with regards to the generated temporal

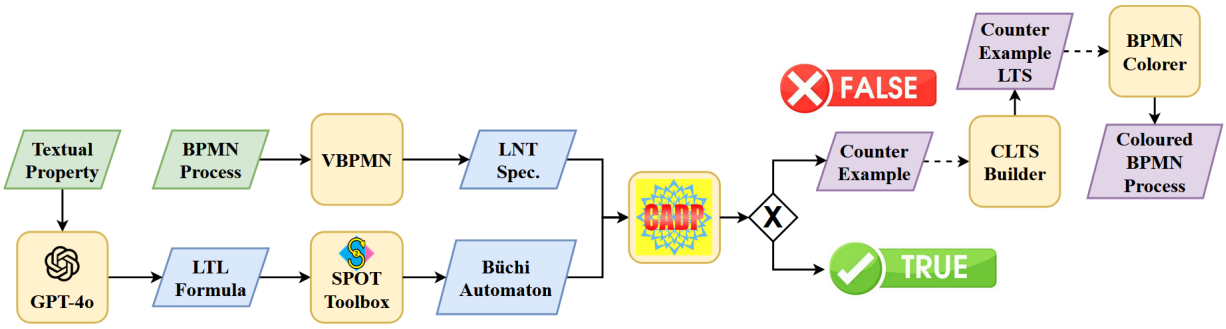


Figure 5.3: Overview of the Toolchain

logic property, written in LTL. It then returns a verdict which is either *True* if the BPMN process satisfies the property, or *False* if the BPMN process violates the property. In case of violation of the property, the verdict also contains a representation of the error, either in the form of a counterexample of the property, or as the set of all counterexamples of the property, or as a coloured version of the original BPMN process. These steps are recalled in Figure 5.3.

5.2.1 Generation of Temporal Logic Property from Textual Description

Based on the experiments conducted in the context of this work, we concluded that LLMs have, for now, a very basic knowledge of temporal logics. Thus, generating an LTL property matching exactly the expectations written by the user in natural language is a challenging task. For this reason, GIVUP is for now recognising nine precise patterns, corresponding to five LTL formulas, on which GPT was trained.

- Pattern 1 associates the textual input

  "Tasks T_1, T_2, ..., T_n must always occur"

  to the LTL property

  $$\varphi_1 = \bigwedge_{i=1}^{n} \mathbf{F}\ T_i$$

  which means that the T_1, T_2, ..., T_n must eventually be executed in one of the paths of the process.

- Pattern 2 associates the textual inputs

  "Tasks T_{n+1}, ..., T_z must follow tasks T_1, T_2, ..., T_n "

  and

  "Tasks T_1, T_2, ..., T_n must be followed by tasks T_{n+1}, ..., T_z"

to the LTL property

$$\varphi_2 = \mathbf{G}\ (\bigvee_{i=1}^{n} T_i \Rightarrow \bigwedge_{j=n+1}^{z} \mathbf{F}\ T_j)$$

which means that, in every state of the system, encountering a $T \in \{T_1, T_2, ..., T_n\}$ must ensure the encountering of the T_{n+1}, ..., T_z in one of the paths of the process.

- Pattern 3 associates the textual inputs

  "Tasks T_1, T_2, ..., T_n must precede tasks T_{n+1}, ..., T_z"

  and

  "Tasks T_{n+1}, ..., T_z must be preceded by tasks T_1, T_2, ..., T_n"

  to the LTL property

$$\varphi_3 = \bigwedge_{i=n+1}^{z} (\mathbf{G}\ \neg T_i\ \vee\ (\bigwedge_{j=1}^{n} \neg T_i\ \mathbf{U}\ T_j))$$

  which means that either the T_{n+1}, ..., T_z never appear in the process, or they do not appear before the T_1, T_2, ..., T_n.

- Pattern 4 associates the textual inputs

  "Tasks T_1, T_2, ..., T_n must not precede tasks T_{n+1}, ..., T_z"

  and

  "Tasks T_{n+1}, ..., T_z must not be preceded by tasks T_1, T_2, ..., T_n"

  to the LTL property

$$\varphi_4 = \bigwedge_{i=n+1}^{z} (\mathbf{G}\ \neg T_i\ \vee\ ((\bigwedge_{j=1}^{n} \neg T_j)\ \mathbf{W}\ T_i))$$

  which means that either the T_{n+1}, ..., T_z never appear in the process, or the T_1, T_2, ..., T_n never appear before them.

- Pattern 5 associates the textual inputs

  "Tasks T_{n+1}, ..., T_z must not not follow tasks T_1, T_2, ..., T_n"

  and

  "Tasks T_1, T_2, ..., T_n must not be followed by tasks T_{n+1}, ..., T_z"

  to the LTL property

$$\varphi_5 = \bigwedge_{i=1}^{n} \mathbf{G}(T_i \Rightarrow \bigwedge_{j=n+1}^{z} \neg\mathbf{F}\ T_j)$$

which means that, in every state of the system, either the T_1, T_2, ..., T_n never appear, or the T_{n+1}, ..., T_z can never appear afterwards.

These patterns were used to fine-tune GPT in order to make it able to generate the correct property in these precise cases. This fine-tuning task was performed on approximately 260 descriptions of properties, with roughly the same number of descriptions (~ 30) for each available pattern.

5.2.2 Verification of the Property

At this stage of the approach, we have a BPMN process, obtained by the approach presented in Chapter 3, and an LTL property exhibiting a desired behaviour of the process. The next step thus consists in model checking the BPMN process with regards to the property in order to assess its validity. However, model checkers do not natively support the BPMN format as input. Thus, the process first has to be transformed into a format compliant with classical model checkers, which is, in this approach, an LTS.

VBPMN [KPS17] is a tool allowing one to translate a BPMN process into LNT [CCG+11], a modern formal specification language that combines traits from process calculi, functional languages, and imperative languages. An LNT specification can then be mapped to its equivalent LTS representation using the CADP toolbox.

As CADP does not natively support the LTL language, the LTL property must be converted into a format that it can understand. This format is a Büchi automaton [Büc66], whose generation is ensured by the SPOT toolbox [DLRC+22]. The Büchi automaton is then converted into an LTS by internal scripts. These internal scripts are also in charge of creating an EXP 2.0 file [Lan05], detailing how to make the synchronous product of the LTSs representing respectively the BPMN process and the LTL property. Finally, these scripts create another script, written in the Script Verification Language (SVL) [GL01], which gives directives to CADP on how to verify the property on the process.

5.2.3 Diagnostics

The verification of the property by CADP leads to a verdict, which is either *True* if the property holds on the given specification, or *False* if the specification violates the property. In the latter case, CADP returns a diagnostic in the form of a counterexample. This counterexample is a trace of the LTS that violates the given property. However, a counterexample is often not sufficient to show precisely the source of the violation of a property.

Example. Let us illustrate the first diagnostic with the BPMN process used in Chapter 4 and the property $\mathbf{G}\ \neg$Send payment request $\vee$ ($\neg$Prepare parcel $\mathbf{W}$ Send payment request) meaning that task Prepare parcel should not happen before task Send payment request. The property is violated in the original BPMN process because both tasks are in parallel, thus their order of execution can not be ensured. In this case, GIVUP returns a counterexample

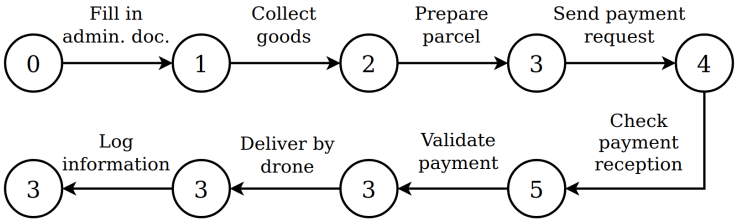


Figure 5.4: Example of Counterexample Returned by GIVUP

which is a trace of the LTS corresponding to the process that violates the property. It is shown in Figure 5.4. As the reader can see, in the counterexample, the task Prepare parcel precedes the task Send payment request.

Counterexample LTS

In addition to the counterexample returned by CADP, GIVUP provides an additional debugging option, called *counterexample LTS* or *CLTS*. This notion, introduced in [BLS17], basically consists in an LTS containing all the traces of the process that violate the property (i.e., the counterexamples), instead of the single one returned by the model checker. With this representation, the user is more likely to find the source of the error, or to understand the global impact that this error has on the model. Although being very helpful to localise the source of the error, the CLTS may remain difficult to interpret or understand for a user that is not familiar with the notion of LTS, due to its lack of similarity with the initial BPMN process.

Example. The CLTS returned by GIVUP according to the previous example is given in Figure 5.5. As the reader can see, this representation is more expressive than a single counterexample. The brown state in sparsed dots describes a decision point after which the property can be either satisfied or violated. If the task Send payment request is executed, the property is necessarily satisfied (reason why state 3 is in green dashed line). On the other hand, if the task Prepare parcel is executed, the property is necessarily violated, whatever happens next.

Coloured BPMN

For users accustomed to BPMN, the CLTS representation, although giving more details on the error than the counterexample, may be insufficiently explanatory. For this reason, we provide a third and last debugging option, called *coloured BPMN process*. This notion, introduced in [NS22], represents directly on a BPMN process the violation and/or the satisfaction of a property by colouring the parts of the model satisfying the property in green, and the ones violating it in red. The coloured process is as syntactically close as possible to the original process, and aims at helping users not familiar with LTSs to understand the reason behind the violation of the property.

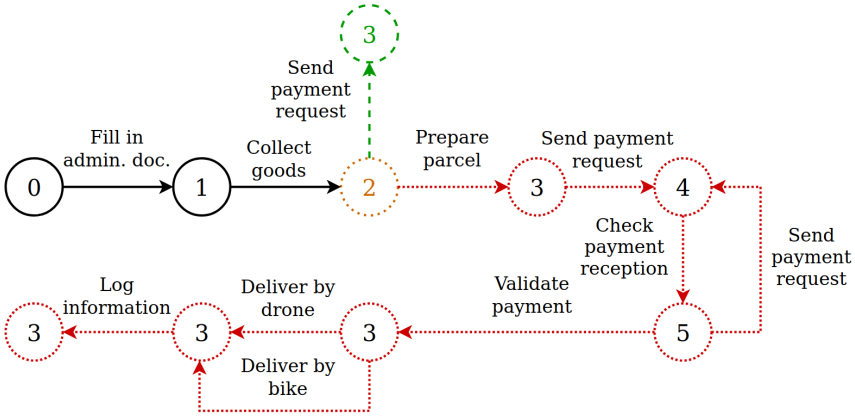


Figure 5.5: Example of CLTS Returned by GIVUP

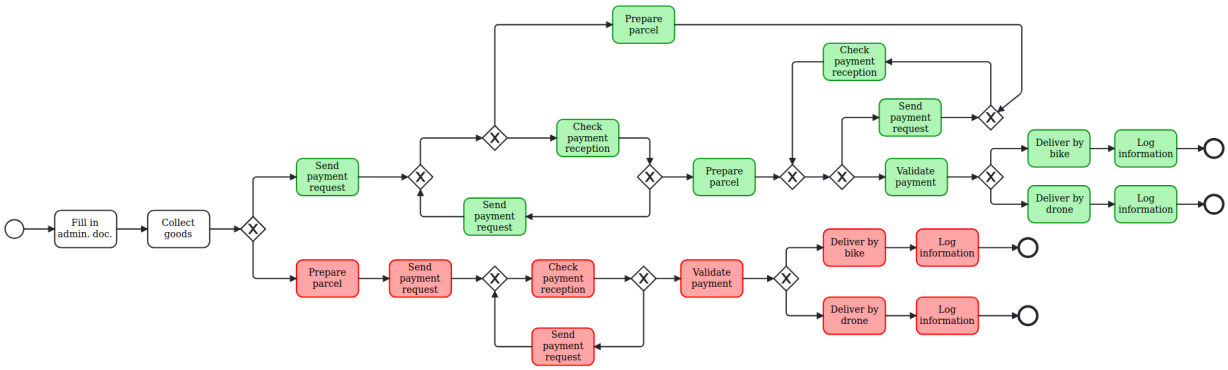


Figure 5.6: Example of Coloured BPMN Returned by GIVUP

Example. Figure 5.6 shows the coloured version of the running example BPMN process, according to the property. As the reader can see, it slightly differs from the original one. To represent the violation of the property, the parallel split gateway following task Collect goods had to be transformed in its semantically equivalent mutually exclusive version, otherwise it could not have been coloured. The fact that one of the parallel branch contains a loop also explains the unusual structure of the green part of the model, as task Prepare parcel can be executed at several possible moments. Finally, it is worth noticing that, unlike in BPMN, LTSs contain labels on their edges. Thus, an LTS representation of a BPMN loop first contains the first executable task of that loop, followed by the loop itself, starting from its second node. Here, it is symbolised by the fact that, in both red and green paths, the first occurrence of task Send payment request is before the loop.

5.2.4 Implementation

The GIVUP tool consists of approximately 20k lines of Java code (12k for the generation of the BPMN process and 8k for the verification), which were embedded in the prototype described in the previous section. The UI of the tool is presented in Figure 5.7. It can

roughly be divided in an upper part and a lower part. Figure 5.7(a), representing the upper part, is composed of 7 core components.

Component 1 is the password field. Usually, LLM-based tools are equipped with an "API key" field, in which the user should input its own API key. However, in our approach, we make use of a fine-tuned version of a GPT model. Such models are accessible only by their owners, requiring their own API key. As hardcoding my own API key would allow a free use of the website, I decided to limit its usage by using a password and quotas. Component 2 is the business process description field. As the reader can see, the user can either generate the business process from a textual description, or upload it if he simply wants to verify it. Similarly to component 2, component 3 is the temporal logic description field. Again, the property can either be generated from a textual description, or uploaded. Then, component 4 submits the request to the server, which is in charge of managing it. It can either contain a BPMN process or a description of it, and a temporal logic property or a description of it. The click on this component triggers component 5 to eventually show the BPMN process or a generated graphical version of its description. Components 6 and 7 are respectively utilised for resetting the whole view, or for downloading the generated process and property.

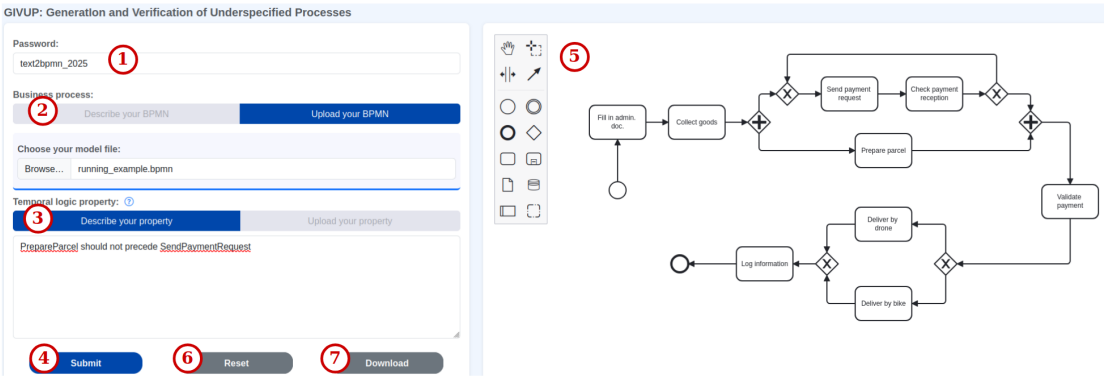
The below part is itself in charge of displaying the information regarding the validity of the property. It is composed of 6 main components and shown in Figure 5.7(b). Here, component 1 is in charge of displaying the information regarding the validity of the property. It is red if the property is violated, and green if it is satisfied. Components 2, 3 and 4 are respectively used to display the different representations of the error, if any. The first one shows the counterexample representation, the second one the CLTS, and the last one the coloured BPMN process. These three representations are displayed in component 5. Finally, the user can indicate its satisfaction of the result by using component 6.

5.2.5 Validation

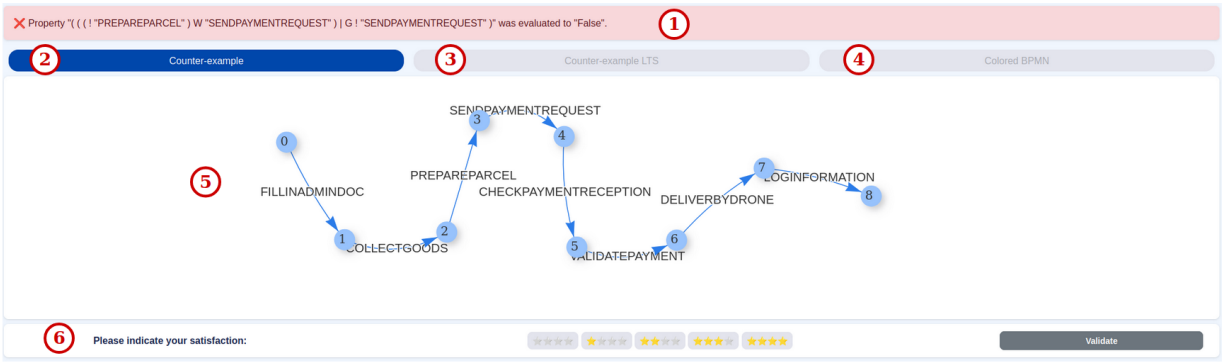
To evaluate the approach, several experiments were conducted. These experiments are divided into two parts: quality of the generated LTL property, and performance of GIVUP.

Quality of the Generated LTL Property

In this approach, the generated LTL property must correspond exactly to the textual description written by the user. Otherwise, the verification step becomes useless, as the behaviour assessed by the model checker is not the expected one. To obtain the desired property, GIVUP currently restricts the user by forcing her/him to choose between the nine well-defined patterns in Section 5.2.1. On these nine patterns, the properties generated from one hundred examples were all correct.



(a) Upper Part of the GIVUP Tool



(b) Lower Part of the GIVUP Tool

Figure 5.7: Screenshots of the GIVUP Tool

Table 5.3: Results of the Performance Experiments Conducted on GIVUP

BPMN Process	States	Trans.	BPMN Gen. Time	Prop. Gen. Time	Model Check. Time	CLTS Cons. Time	BPMN Colo. Time
Evisa Application [Sal22]	30	31	1.67s	1.43s	4.35s	3.12s	613ms
Patient Diag. [BZOW19]	38	40	2.71s	1.14s	4.57s	3.29s	661ms
Employee Recrui. [FSZ21]	39	40	1.89s	1.33s	4.57s	3.22s	830ms
Employee Hir. [DRS18a]	78	105	2.17s	2.86s	4.34s	3.22s	758ms
Perishable Goods [VTS22]	108	150	2.22s	1.1s	4.78s	3.31s	978ms
Account Open. [NS22]	304	657	2.31s	1.23s	4.56s	4.23s	1.42s
Hardware Retail. [FSZ22]	373	819	2.56s	1.1s	4.53s	3.13s	764ms
Online Shipp. [KPS19]	375	765	2.78s	1.07s	5.12s	3.27s	1.7s
Handcrafted 1	279k	1.63m	3.25s	1.07s	8s	14.6s	17.2s
Handcrafted 2	1.67m	11m	1.79s	1.27s	23.9s	5.42m	24m
Handcrafted 3	10m	75m	1.67s	1.18s	3.05m	>1h	>1h
Handcrafted 4	60m	503m	2.08s	867ms	27.7m	>1h	>1h
Handcrafted 5	362m	3.32b	2.31s	1.85s	>1h	>1h	>1h

Performances of GIVUP

The second part of these experiments aimed at assessing the scalability of GIVUP. They were conducted on both real-world examples coming from the literature, and handcrafted examples. Table 5.3 summarises the results. Column 1 provides the origin of the process, Columns 2 and 3 give details about the LTS corresponding to the specification (i.e., number of states and transitions of the LTS corresponding to the BPMN process), and Columns 4, 5, 6, 7 and 8 present the time taken by each major step of the approach to complete (i.e., generation of the BPMN process, generation of the LTL property, model checking of the property, construction of the CLTS, and colouration of the BPMN process).

These experiments show that the total execution time including the non-mandatory steps (CLTS construction and BPMN colouration) never exceeds 15s for real-world examples[3]. The mandatory steps are themselves executed in at most 10s for real-world examples. For the model checking step, the first limitations appear for processes containing approximately 10 million states, as CADP takes a few minutes to compute the result. This limit corresponds to the well-known state explosion issue that CADP (and all model checkers) suffers from. For the CLTS generation, the limit is reached earlier. Indeed, the construction of the CLTS requires the computation of all the counterexamples of the property, which implies repeating the model checking step multiple times, thus summing its execution time. Finally, to be able to colour the BPMN process, an intermediate step called *unfolding* is required. Unfolding consists in duplicating each node of the BPMN process having more

[3]The reader testing GIVUP may experience longer execution times due to the distance from the server hosting it.

than one incoming flow until no such node exists in the process. Consequently, this step increases exponentially the number of nodes of the process, making the colouring process longer than the CLTS generation. However, it is worth noting that, in practice, the LTS model of a BPMN process rarely exceeds a few thousand states, and therefore almost never suffers from any of the aforementioned issues, thus ensuring a short execution time.

5.3 Refactoring

In this section, we will focus on the tools built around the refactoring approach, and their corresponding experiments. There are three tools, each corresponding to one of the approaches presented in Chapter 4, which roughly consist of 25k lines of Java code. The first one deals with the fixed durations approach, and thus performs the refactoring in a one-shot manner by applying the successive steps presented in Section 4.5. The second one deals with the non-fixed durations step-by-step approach, and thus performs several successive refactoring steps, each of which is validated by the user using the tool, as described in Section 4.6. The last one deals with multiple optimisation criteria, and is rather similar to the second one, except that the user is no longer involved in the loop. It corresponds to the description made in Section 4.7.

5.3.1 Fixed Durations Approach

For the fixed durations approach, the tool was written in Java and consists of approximately 10k lines of code. It has been tested on various handcrafted and real-world examples found in the literature. The experiments allowed us to evaluate our approach both in terms of usefulness and performance, by considering the gain of the optimised processes in terms of AET, and the time taken by the tool to execute. The number of instances for each considered process varies between 20 and 100.

Table 5.4 summarises these experiments. Column 1 gives the name of the process and its origin. Columns 2, 3 & 4 show several characteristics of the process (number of nodes, flows, types of resources, replicas of resources, IAT). Columns 5 & 6 provide respectively the AET of the initial process and of the optimised process. Column 7 shows the gain that was obtained by optimising the process. Column 8 states whether the available pool of resources is sufficient to execute the optimal version of the process or not. Column 9 gives the time taken by the tool to execute.

The results can be split into two parts: the processes having enough resources to execute the optimal version of the process, and the others. In the first case, the AET of the generated process is optimal, and is generally a significant improvement of the initial one (up to 46% for the first process). A lower gain only indicates that the initial process was already syntactically close to its optimal form, not that the approach does not perform well. In the second case, the gain is lower than in the first case (up to 15.5%), due to the decrease of parallelism induced by the sequencing of some tasks. Overall, the tool executes

Table 5.4: Results of the Experiments Conducted for the Fixed Durations Refactoring Approach

BPMN Process	Nodes/ Flows	Types/ Number of Res.	IAT	Init. AET (UT)	Final AET (UT)	Gain (%)	Min. Res.	Time (ms)
Perish. Goods [VTS22]	24/26	9/17	6	26	14	46.2	✓	563
Employee Hiring [DRS18a]	19/21	7/12	3	30	18	40.0	✓	607
Trip Organisation [DS22]	11/11	6/11	7	41	28	31.7	✓	588
Patient Diag. [BZOW19]	14/15	4/12	20	61	46	24.6	✓	624
Shipment Process [FSZ22]	16/18	5/10	5	46	42	8.70	✓	531
Evisa Application [Sal22]	11/11	3/7	5	84	71	15.5	✗	797
Employee Recruit. [FSZ21]	14/14	7/11	5	92	80	13.0	✗	873
Account Opening [NS22]	22/25	6/17	8	67	63	5.97	✗	732
Goods Delivery [DFR+22]	11/12	6/16	1	78	77	1.28	✗	696

in less than 1s on real-world processes, which is satisfactory as this approach is executed at design time.

5.3.2 Non-fixed Durations Step-by-Step Approach

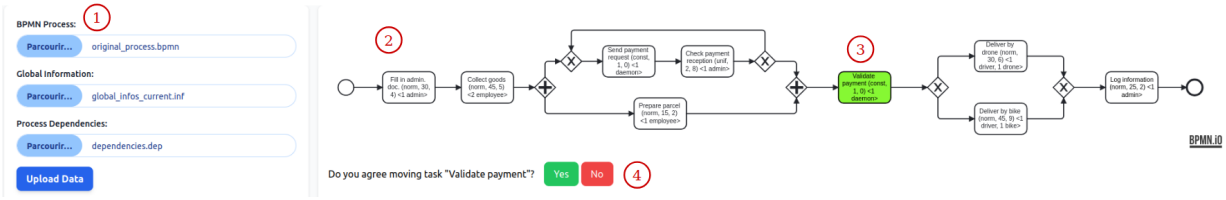
This subsection gives details about the tool support of the non-fixed durations single objective approach, and describes the experiments conducted to evaluate and validate it.

Tool Support

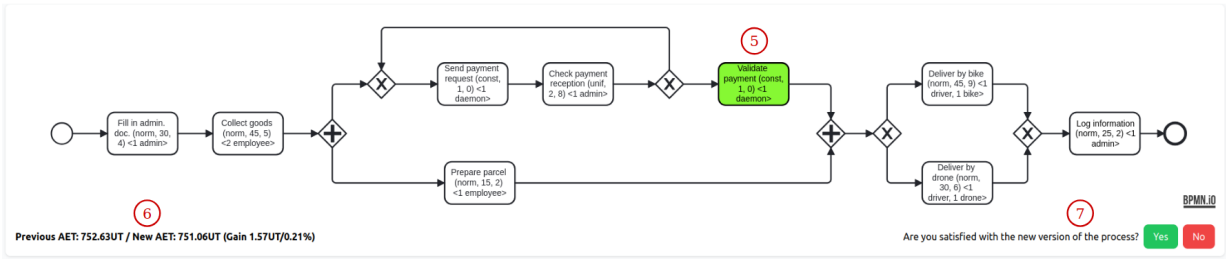
The approach has been fully implemented as a tool written in Java which consists of approximately 15k lines of code. For distribution purposes, it was embedded in a JAR file that executes in the backend of a NodeJS server running locally. It is freely available online[4]. It has been fully tested and evaluated on several real-world examples and hundreds of handcrafted examples. Figure 5.8 shows two screenshots of the frontend of the web server that the user can use.

In Figure 5.8(a), the user has uploaded her/his BPMN process, the global information (IAT, available resources, number of instances) and the dependencies of the process (step 1). The BPMN process is then displayed using the bpmn.io API[5] (step 2), and the tool proposes the first task to move to the user, i.e., task Validate payment (step 3). It also asks the user whether (s)he agrees to move this task or not (step 4). If the user accepts, the tool then computes the best position of this task (step 5), and displays it on the screen, as shown in Figure 5.8(b). Otherwise, it proposes another task until the user accepts to move a task. Here, as the user accepts, the resulting process is shown, and the gain of this new process

[4] https://quentinnivon.github.io/pages/softwares.html
[5] https://bpmn.io/



(a) Tool Proposing the Task to Move to the User



(b) Tool Showing the Generated Process to the User

Figure 5.8: Screenshots of the Tool Support of the Approach

compared to the previous one is displayed (step 6). The user is finally asked to accept or decline this new process (step 7), which triggers, in both cases, a new iteration.

Experiments

The experiments described in this section aim at assessing the performance of the tool, as well as the quality of the process returned by the score-based heuristic in terms of AET. To do so, the results obtained by the heuristic are compared to the ones obtained by the full exploration on several BPMN processes executed 100 times with multiple resources, both in terms of AET and computation time of the tool. The results of this analysis are given in Table 5.5. The columns present the considered BPMN processes with their name and their origin. The lines are separated into three blocks. Block 1 contains the characteristics of the original process, namely, its number of tasks, choice structures, dependencies, and its AET. Block 2 provides the results of the refactoring operation obtained by using the heuristic presented in Section 4.6.2. The information is given as the AET of the refactored process, the gain (in percents) compared to the original process, the time taken by the refactoring operation to complete, and the time taken by each step to complete. Finally, block 3 gives the AET and the gain of the optimal version of the original process obtained with a full exploration of the state space. The star symbol (*) in line 5 (AET) highlights cases where the fourth generation pattern (Definition 4.18) was (at least partially) discarded due to the important number of generated processes. It is worth noting that, as these results aim at assessing the performance of the tool and the quality of the heuristic, they were performed under the assumption that the user accepts all the steps that were proposed to her/him.

		Evisa App. [Sal22]	Empl. Rec. [FSZ21]	Patient Diag. [6]	Empl. Hir. [7]	Acc. Op. [NS22]	Per. Goods [VTS22]	Online Ship. [8]	Hand-Crafted 1	Hand-Crafted 2a	Hand-Crafted 2b
Characs.	Tasks	9	10	8	11	15	16	24	26	51	51
	$\langle\!\langle \times \rangle\!\rangle_C$	1	1	2	2	2	2	3	4	1	1
	Deps.	3	9	3	7	7	10	27	43	43	23
	AET	36.1	30.9	67.2	24.7	51.9	15	85.9	232	323	323
Heuristic	AET	20	21.4	61.6	19	40.9	13.2	70.3	145*	244*	182*
	Gain	44.6%	30.7%	8.33%	23.1%	21.2%	12.0%	18.2%	37.5%	24.5%	43.7%
	Time	6.21s	32s	5s	26s	1.25m	14s	1.97m	6.37m	58m	2.12h
	μ_{time}	0.88s	0.32s	0.56s	2.36s	5s	1.75s	4.9s	15s	1.14m	2.54m
Full	AET	17.1	20.4	60.4	17.8	34.2	13.2	69.3	122	183	99
	Gain	52.6%	34.0%	10.1%	27.9%	34.1%	12.0%	19.3%	47.4%	43.3%	69.3%
	Time	17.0m	7.38m	11.2s	43.1h	26.8h	1.34h	>14d	1.7d	>14d	>14d

Table 5.5: Results of the Experiments Conducted for the Non-fixed Durations Step-by-step Refactoring Approach

As the reader can see, the heuristic performs very well on real-world examples, as it returns processes with an AET close to the optimal one (and even the optimal one for example 6). The worst result obtained by the heuristic on real-world examples is for example 5, for which the gain of the heuristic is 33.5% worse than the gain of the full exploration. On average, the heuristic performs only 13.9% less well than the full exploration, while being much faster. The approach also performs in reasonable time on real-world examples, with a time per step reaching 5 seconds at most. On the first handcrafted example, containing 26 tasks, the approach still obtains a non-negligible gain of 37.5% in a reasonable time per step (15s). Finally, the approach shows some limitations on the last two handcrafted examples, which are two variants of a 51 tasks process with a different number of dependencies, and for which the execution time per step is no longer acceptable in a real-time context (more than 1m). It is worth noting that, as expected, the number of dependencies and the size of the process play an important role in the computational time of the approach.

5.3.3 Multi-Objectives Approach

This section presents the experiments that were conducted in the scope of the multi-objectives refactoring approach presented in Section 4.7. This approach partly relies on the jMetal framework [DN11]. jMetal is an open source Java-based framework for multi-

[6] [BZOW19] was placed here to ensure a proper rendering of the table.
[7] [DRS18a] was placed here to ensure a proper rendering of the table.
[8] [DRS19] was placed here to ensure a proper rendering of the table.

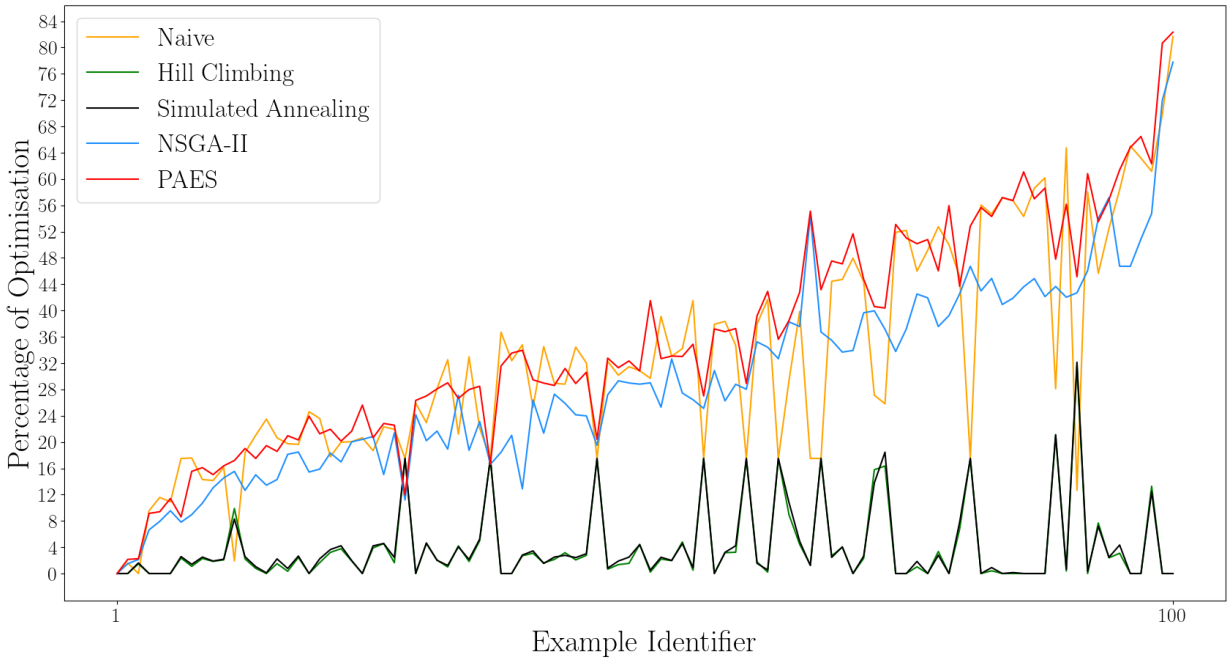


Figure 5.9: Comparison of the Selected Algorithms per Example

objective optimisation with metaheuristics. It includes a wide set of resources, including state-of-the-art multi-objective algorithms, solution encodings, benchmark problems, quality indicators, and utilities for performing experimental studies. The whole approach has been fully implemented and consists of roughly 15k lines of Java code (2.5k for the jMetal implementation, 7k for the refactoring approach, 1.5k for the simulation and the remaining 4k for the handling of the BPMN notation). The approach was also tested on a hundred of handcrafted examples and a dozen of real-world ones, whose results will be presented in the next sections.

Process Refactoring Algorithms Evaluation

In this section, we compare the evolutionary algorithms presented in Section 4.7.4. As a witness of quality, we also compare them to the fast algorithm presented in Section 4.7.4, that is expected to provide quickly a (less valuable) result. These algorithms were compared on both synthetic and real-world examples.

Synthetic Examples.

The first set of experiments conducted consisted in assessing whether some of these algorithms were more suitable than the others to the refactoring problem, i.e., if they were computing better solutions. In an attempt to compare the algorithms as fairly as possible, all the algorithms were executed with a maximum execution time as stopping criterion. Moreover, these experiments allowed us to analyse the behavior of these algorithms over time, and conclude on a time bound giving the best trade-off between quality of the re-

sulting process and execution time.

Even though dominance is the most common way of comparing two solutions, it does not fit well in our case. Indeed, despite the fact that the usage of a specific resource is relevant, clearly the AET or the cost may have a higher impact on the process. Indeed, we observed that "better" solutions were being discarded because they were not dominant, even though they had a better AET or cost. So, we decided to use a weighted sum of the criteria to compare the solutions. The following weights were used: 30% for AET, 20% for cost, 20% for R_1 usage, 20% for R_2 usage, and 10% for R_3 usage. As we will see below, the use of this weighted sum will also allow us to compare the results of the algorithms with the original process, and to compute the percentage of optimisation of the resulting process with respect to the original one.

These experiments were conducted on 100 randomly generated examples and repeated 20 times for stability. The results are shown in Figure 5.9.[9] The figure shows the average optimisation percentage per algorithm. The optimisation percentage of a process is the weighted sum of the optimisation percentages of all of its optimisation criteria.

We can observe that the algorithms obtaining the worst performances are hill climbing and simulated annealing, which do not explore the state space as efficiently as the others. Indeed, they provide either no improvement or a very small one. The best algorithms are NSGA-II and PAES, which both achieve a good optimisation percentage. The optimisation fluctuates between 0% and 80%, depending on the example. The fast algorithm performs surprisingly good, outperforming the hill climbing and the simulated annealing algorithms, but not reaching the performance of NSGA-II and PAES in most of the cases. It is worth noting that the fast algorithm still outperforms both NSGA-II and PAES on some examples, which is a good indicator that the refactoring patterns are effective in improving the processes. In fact, there should be no reason why a task should be moved more than once. The key seems to be the order in which they are moved, which may be unrelevant in some cases, even though crucial in others. What is very significant is the time consumed by the different algorithms. Even though we will present execution times for the examples in the next section, the fast algorithm executes in a few seconds, while the other algorithms take several minutes to complete. As already pointed out, the execution time of the four remaining algorithms was bounded to 5 minutes.

Real-World Processes Evaluation.

The second part of these experiments consists in evaluating the quality of the approach on examples coming from the literature. These experiments were conducted on eight examples, including the running example of Section 4.6. For each example, several criteria of optimisation were taken into account, namely, the AET, the cost, and the average usage of three resources (named R_1, R_2, and R_3 for homogeneity).

[9]The examples in Figure 5.9 were sorted from lowest optimisation percentage to greatest optimisation percentage to facilitate their interpretation.

Table 5.6: Results of the Experiments Conducted on the Multi-objectives Refactoring Approach

		Goods Deliv. (Ex. 4.6)	Evisa App. [Sal22]	Empl. Rec. [FSZ21]	Patient Diag. [10]	Empl. Hiring* [11]	Acc. Open.* [NS22]	Perish. Goods* [12]	Online Ship. [DRS19]
Characteristics	Tasks	7	8	10	8	10	12	13	24
	◈$_C$/✖$_L$/✚	1/0/1	1/0/0	1/0/0	2/0/0	1/1/0	2/1/1	2/0/1	3/1/3
	Deps.	9	3	9	3	12	7	20	27
	IAT (UT)	3	5	5	20	3	12	6	25
	$R_1/R_2/R_3$	1/2/4	5/2/3	2/1/2	4/3/5	5/5/4	3/2/1	2/3/1	2/2/3
Original	AET (UT)	750.8	159.6	25.0	58.7	575.6	67.8	20.8	98.9
	Cost	691.4k	85k	174k	652.5k	542.4k	177.5k	243.8k	1.131m
	R_1 usg.	97.7%	30.0%	67.0%	33.6%	13.9%	80.7	81.4%	70.4%
	R_2 usg.	52.6%	97.9%	19.1%	63.2%	4.19%	19.6	19.0%	74.9%
	R_3 usg.	56.4%	23.9%	95.7%	12.3%	4.38%	50.8	16.3%	34.3%
Optimal	AET (UT)	520.8	130.1	16.8	46.9	512.5	47.7	18.4	N/A
	Cost	696.8k	82.4k	171.1k	650k	493.4k	172.9k	242.9k	N/A
	R_1 usg.	96.6%	30.2%	68.1%	33.3%	14.0%	79.1%	80.7%	N/A
	R_2 usg.	52.1%	98.3%	19.5%	62.7%	4.30%	19.8%	19.2%	N/A
	R_3 usg.	55.0%	23.0%	97.3%	12.3%	4.86%	50.4%	16.3%	N/A
	Gain	34.2%	25.3%	31.0%	22.9%	19.5%	31.5%	13.0%	N/A
	Time	5.74d	17.0m	7.38m	11.22s	43.1h	26.8h	1.34h	>14d
Fast	AET (UT)	528.5	153.6	17.0	50.9	571.7	50.6	18.5	84.1
	Cost	691.4k	84.8k	170.7k	650.9k	539.7k	174.1k	242.9k	1.124m
	Gain	30.2%	3.73%	30.4%	13.3%	1.07%	24.5%	11.3%	**13.6%**
	Time	7.8s	2.43s	5.80s	2.63s	3.35s	2.46s	6.86s	55.7s
Hill Clim.	AET (UT)	716.7	142.3	24.5	57.7	529.3	62.5	20.4	98.1
	Cost	687.1	82.3k	174.2k	652.7k	511k	176.7k	243.7k	1.132m
	Gain	5.57%	15.5%	2.44%	7.34%	13.1%	13.7%	2.79%	2.44%
Simu. A.	AET (UT)	717.2	141.9	24.3	57.7	523.4	62.2	20.4	97.6
	Cost	686.5k	82.5k	173.9k	652.3k	508.5k	176.8k	243.8k	1.131m
	Gain	6.01%	15.8%	2.83%	7.92%	14.3%	13.5%	3.4%	2.67%
PAES	AET (UT)	526.5	142.0	16.8	46.3	516.6	44.5	18.2	86.5
	Cost	696.4k	82.2k	170.9k	649.5k	503.8k	173k	242.9k	1.126m
	Gain	**31.7%**	16.4%	**31.4%**	**24.6%**	**16.2%**	**33.9%**	**13.0%**	12.8%
NSGA-II	AET (UT)	569.6	140.7	17.7	47.9	520.9	46.3	18.7	87.6
	Cost	691.6k	82.4k	171.5k	649.9k	505.9k	173.7k	243.1k	1.127m
	Gain	25.5%	**17.4%**	28.3%	**24.4%**	13.7%	**34.7%**	11.7%	12.3%

The table shows the results of running the fast, the hill climbing, the simulated annealing, the NSGA-II, and the PAES algorithms on these eight processes. As a reference, the results of a full exploration algorithm are also shown in Table 5.6 (Optimal). The table is organised as follows: Each column corresponds to one of the 8 examples used, with the running example in column 1 (Goods Deliv.). Notice that some examples are followed by an asterisk (columns 5–7), indicating that these examples have been simplified in order to make the computation of their optimal version possible in a reasonable time. To illustrate the time complexity, the example in column 8 (Online Ship.) was left unaltered and ran for 2 weeks before the computations were halted without yielding a result.

The rows are split in 8 blocks.

- Block 1 (Description) contains indicators on some of the structural features of the considered process to wrasp an idea of its complexity, namely, its number of tasks, its number of choice, parallel and loop structures, its number of dependencies, and the parameters that were used to simulate it, namely, the IAT, and the number of available replicas of each resource R_1, R_2, and R_3. The number of instances (100), being identical for each process, is not displayed.
- Block 2 (Original) presents the metrics obtained by the original process for each considered optimisation criterion. These metrics are shared with the 6 remaining blocks, where the results for each of the four exploration algorithms are shown. Please, note that since the cost is calculated as the cost of the resources used along the execution, and therefore depends on the execution time and resource usage, resource usage values are not shown for the different algorithms to allow the table to fit into one page.
- In block 3 (Optimal), the metrics of the optimal process are given, along with the percentage of optimisation compared to the original process, named *Gain*, and the time taken by the algorithm to compute the solution.
- Blocks 4 to 8 (Naive, Hill Climbing, Simul. Anneal., PAES, and NSGAII) show the results obtained by the corresponding algorithms. The last four blocks were executed using their jMetal implementation, with a *soft bound* of 5 minutes, which means that the algorithm is not immediately killed once the bound is reached, but it is allowed to terminate its current iteration.

We can observe several interesting results from this table. First, one can see that computing the optimal process using a full exploration of the state space is not applicable in most cases, as the time taken by the computation exceeds days on rather small examples, and even weeks for the last example. Another interesting thing that one can observe is that the fast algorithm, although behaving quite simply, performs well in some cases (examples 1, 3,

[10] [BZOW19] was placed here to ensure a proper rendering of the table.
[11] [DRS18a] was placed here to ensure a proper rendering of the table.
[12] [VTS22] was placed here to ensure a proper rendering of the table.

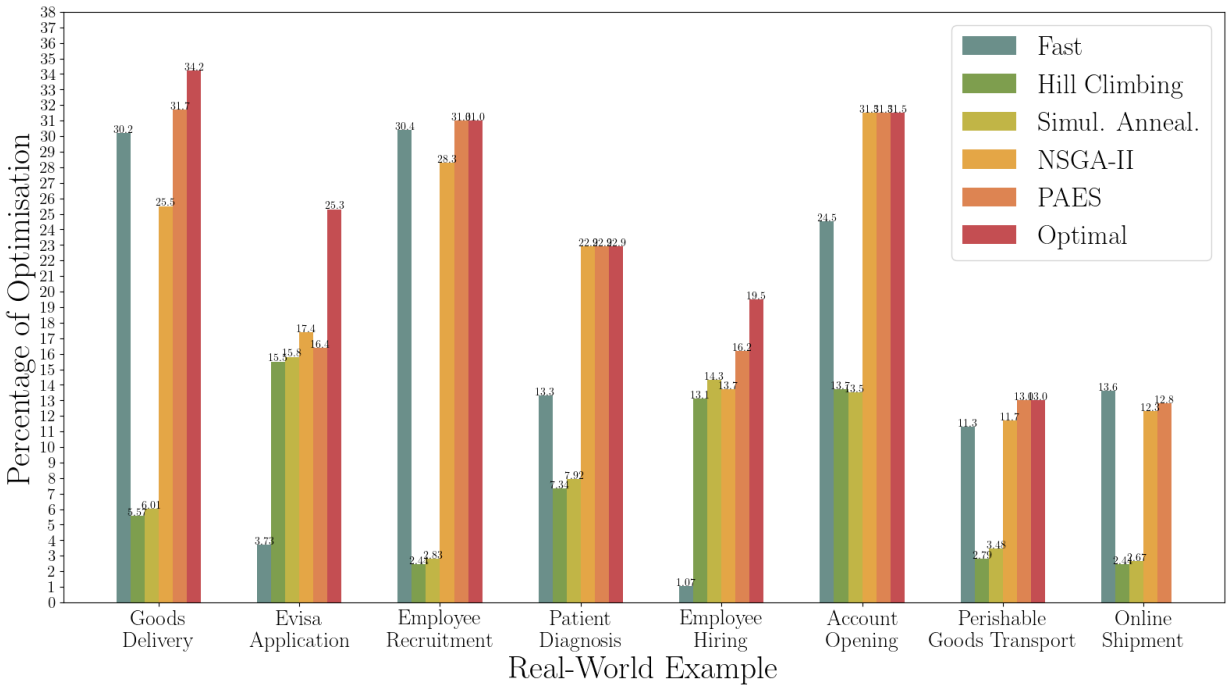


Figure 5.10: Distance to the Optimal Solution

6, 7, and 8), and even better than the other algorithms for example 8. Remarkably, the fast algorithm is able to get such results in a very short time: less than 6 seconds in all cases but the one in column 8. Even though it takes 55.7 seconds for this example, this is still a very short time compared to the other algorithms, which is even more impressive as it managed to get a better gain than all the others. However, due to its inherent simplicity, it cannot avoid all the pitfalls (local maxima, bad task positioning, ...), and sometimes performs badly, as in examples 2 and 5. Regarding NSGA-II and PAES, they both obtain very good results in all the cases, and are even able to reach the optimal in several cases (NSGA-II in examples 4 and 6, and PAES in 3, 4, 6 and 7).[13] As for the synthetic examples, hill climbing and simulated annealing behave quite poorly in most cases, but are useful to set a reference value against which the other algorithms can be compared. The distance to the optimal process is summarised in Figure 5.10, in which the good performances of NSGA-II and PAES are even clearer.

5.4 Conclusion

In this chapter, we have presented the various tools that were developed to support the approaches detailed in the previous chapters of this manuscript. In total, these tools

[13]Note that obtaining a gain over the optimal is only possible because the simulation is performed only once for the optimal process, whilst the values shown for the other algorithms are averages of all the experiments carried out.

represent approximately 45k lines of Java code (the lack of precision coming from the code shared between the different refactoring approaches). Each of these tools was used to make experiments and empirical verifications on the feasibility, usability, and quality of the proposed approaches. For each of them, the experiments were successful. Currently[14], some of these tools are available online[15]. The GIVUP tool is freely usable online, hosted on a server, while the step-by-step refactoring tool is downloadable online, and has to be installed on the user's machine beforehand. A particular attention was paid to the packaging and the distribution of these two tools, as they both require a lot of interactions with their users. The remaining tools, despite being developed with the same concern of quality than the others, were not wrapped inside a UI, but remain usable through the command line.

[14]January 23, 2026.

[15]`https://quentinnivon.github.io/pages/softwares.html`

Chapter 6

Related Work

"It is a narrow mind which cannot look at a subject from various points of view."

George Eliot

Contents

This chapter presents several approaches and ideas that were beneficial for the thinking and the construction of this thesis. It will mostly address the challenges encountered when trying to model business processes from natural language descriptions, but also how recent techniques can be leveraged to generate temporal logic formulas from text. Then, it will give an overview of what is meant by *refactoring* in the existing literature, and compare this optimisation method to other well-known ones.

6.1 Modelling

Modelling businesses processes efficiently while providing strong guarantees on the generated process has been an exciting challenge since the rise of business model notations. The reason behind this keen interest lies in the benefits of optimising the construction of such processes. First of all, designing a business process is a tedious and time-consuming task, mostly because it has to be done manually, using the classical graphical tools developed for that purpose. Second, writing such processes usually requires some knowledge of the notation, limiting the number of capable users in a company. Lastly, the vast majority of existing design tools does not provide any checks to assess the (syntactical and semantical) validity of the designed processes. These three reasons mostly suffice to give insights about the craze around automated process generation since the origin of workflow-based notations. Another topic of interest in this thesis is to provide the ability for non-expert users to generate temporal logic properties. Interestingly, there were not many works tackling

this problem in the literature before 2020, which is remembered for the explosion of large language models. Several reasons, such as the inherent complexity of temporal logics, and the existence of patterns for many classical applications, may have explained this disinterest. However, the appearance of LLMs reshuffled the cards and several researchers saw an opportunity in their usage.

6.1.1 Generation of BPMN

The World Before LLMs

In [FMP11], the authors provide a fully implemented technique to generate business processes from textual descriptions, based on natural language processing. According to their study, they put the accent on four broad categories of issues that they tried to solve, namely, *syntactic leeway*, which highlights a mismatch between the syntax of a text and its semantics, *atomicity*, which deals with issues regarding the mapping between phrases and activities, *relevance*, which is in charge of verifying the interest of keeping a given part of the text to generate the process, and *referencing*, which addresses the issue of preserving the connection between words, sentences, and the global meaning of the text. They take advantage of the Stanford Parser [dMMM06] to break down sentences into smaller groups, allowing them to obtain information about the activities, the actors, and the flows relating them. Then, they use some text level analysis based, again, on the Stanford Parser, and on WordNet [Mil95], coupled to a homemade anaphora resolution algorithm. Based on these algorithms, they are able to determine some relations between the elements of the text. They also extract the eventual conditional structures described in the text, based on lists of words characterising the possible gateways behaviours. They finally generate a BPMN process out of all the stored information, and return it to the user. Six years later, the author of [SV17] revisited the paper by proposing new ideas to mitigate the limitations stated by the original authors, but did not provide a concrete solution for them.

The approach presented in [HKW18] aims at generating business processes from natural language descriptions. Prior to the apparition of LLMs, extracting structured, valuable information from a textual description required alternative mechanisms. In this approach, the authors present a spreadsheet-based approach, in which the original description is analysed and used as a basis to create a spreadsheet composed of the elements of the process, that are, the names of the activities, their order, their condition of execution, and the actors performing them. To obtain this intermediate representation, they first extract the actors, or participants, from the text. To do so, they rely on a line-by-line analysis of the description, where a sentence is mapped to a syntax tree representation highlighting the grammatical type of each word composing it. Then, if the analysis returns one or several words, they are compared against a list of plausible words, and kept if they belong to this list. In a second step, they make use of subject-verb-object extraction techniques to retrieve the names of the activities. Finally, they perform a last analysis of the description to find keywords indicating the existence of possible gateways. For instance, the “if [...]

else" construct will lead to the creation of an exclusive gateway. Once the spreadsheet is complete, they make use of a Python library to generate the corresponding BPMN diagram. The authors conclude the paper with a case study illustrating the approach on an example. This approach provides interesting results on the example presented by the authors. However, as all pre-LLMs approaches, it suffers several limitations. Notably, the description requires to be well-structured, and to contain the right keywords to be interpreted correctly. Also, the authors do not provide support for loops, or complex structures such as unbalanced workflows.

The Sketch Miner tool [ISP20] combines notes taken in constrained natural language with process mining techniques to automatically produce BPMN diagrams in real-time. To do so, the authors designed a Domain-Specific Language (DSL), along with its corresponding grammar, in charge of capturing the largest subset of the BPMN syntax, while using a limited number of textual constructs, in order to make it easy to understand, learn, and remember. The user must then write a textual description compliant with the grammar supported by the DSL. A DSL parser is then in charge of extracting traces from this description. These traces are then fed to a process mining algorithm, which is in charge of (re)generating the structure of the BPMN process. This raw representation is finally enriched by roles, task names, and events, that are extracted directly from the original descriptions, using annotations supported by the grammar. The process is finally rendered and returned to the user. The authors of this approach proposed a clever solution for solving the problem of business process generation from natural language. However, similarly to the previously presented approaches, their approach required a structured input, along with a deep knowledge of the designed grammar.

In [FSZ21], the authors present an approach aiming at modelling business processes in a semi-automated way. This approach takes as input a set of *partial orders*, which is a set of couples of dependent tasks. These couples must be compliant with two rules for the corresponding BPMN process to be generated. Then, the authors introduce a notion aiming at representing a business process in a simple way, called *abstract graph*. This abstract graph is a representation of a business process without any explicit control-flow element. Once this abstract graph is generated from the partial orders, it is given to the user who can refine it by, for instance, adding a new node to the graph in order to mimic an exclusive split gateway. Finally, when the user is done modifying this abstract graph, it is transformed to BPMN using a dedicated algorithm. The authors also provide a tool for this approach, presented in [CFSZ22]. Contrarily to our approach, the one presented by the authors is not fully automated, and does not provide support for loops or unbalanced gateways.

First Insights of LLMs Usage

In [BDG22], the authors make use of *in-context learning* [Bro20] and *pre-trained language models* [QSX+20, WLW+23] to extract structured information from a textual description of a business process. This structured information consist of *entities* (i.e., tasks) and

participants, i.e., the actor performing the task. To perform this extraction, they rely on a series of exchanges with the PLM (here GPT-3) which ends when all the desired elements and the relationships between them have been retrieved. An exchange, or prompt, consists of four elements: (i) a question regarding the elements of the description that GPT should focus on, (ii) *contextual knowledge* which provides information to GPT about the BPMN language, (iii) examples of potential answers to eventual questions, and (iv) the textual description of the process-to-be. Once it receives this crafted prompt, GPT "thinks", and returns a detailed answer. Based on this answer, and on its importance with regards to the next prompt (i.e., does it contain necessary information to ask the next question?), the next prompt is crafted, and sent to GPT. Following this schema, GPT is prompted repeatedly until obtaining all the desired information. The authors tested their approach on 7 descriptions coming from the PET dataset [BvdAD+22], and obtained promising results regarding the extraction of such elements in terms of precision, recall, and F1-score. However, their approach is for now limited to the extraction of tasks, actors, and basic relationships between tasks (e.g., is a task before another?). For instance, the control flow of the process remains unknown. Moreover, the approach does not provide experimental data regarding the total time required to extract the information of a process. Hence, it is hard to assess the scalability of the approach for larger descriptions. Finally, the approach is highly dependent on human interaction, and cannot automatically generate a BPMN process.

The authors of [FFK23] provide some preliminary results on the capabilities of (Chat)GPT to generate models based on a precise description of it. They focus on four different types of models, namely, *entity-relationship diagrams* [Che76], *business processes diagrams*, *UML class diagrams* [BRJ05], and *Heraklit models* [FR21]. To increase the quality of the result returned by GPT, they rely on two different techniques. First, they use *prompting* to provide some key information about the targeted model to GPT, which basically gives it some contextual knowledge of the situation. Then, they created their own textual representation of each model, using the JSON format. This representation has the main benefit of reducing the inherent complexity of each of these four models, which are mostly originally written in XML, an unnecessarily complex language decreasing the likelihood of GPT to generate a valuable result. Then, they prompt GPT with (i) the full, detailed explanation of the model, (ii) the complete description of the expected output format, (iii) the textual representation of the model, written in natural language, and (iv) a complete scenario detailing the expected behaviour of GPT. As a response, they obtain from GPT a representation of the model that is valid with regards to the provided metamodel. Let us now restrict the focus to BPMN. For this representation, the authors provide support for tasks, parallel gateways, and exclusive gateways for which splits must have exactly one incoming flow and two outgoing flows. Loops are not (yet) handled by the approach. For a short textual description of a BPMN process, the results are promising as they are able to produce a JSON representation of the BPMN process that is conform to the textual description. However, they do not provide further experiments on other (more complex) processes.

In [KBK+23], the authors present the concept of *conversational process modelling*, which "describes the process of creating and improving process models and process descriptions based on the iterative exchange of questions/answers between domain experts and chatbots". They apply this concept to four application scenarios related to the lifecycle of BPMN processes, for which they identify the key usages for which a chatbot could be helpful. The first one, named *process discovery*, consists in generating a BPMN process from a textual description of the requirements, which is commonly performed by experts. For this scenario, they highlight three key usages of a chatbot: (i) gathering of process descriptions, (ii) production of the process model from the description, and (iii) assessment of the process quality. The second one, named *process analysis*, concerns quantitative and qualitative assessments of process models. For qualitative analysis, the key usage of a chatbot could be to assess the validity of qualitative assessments, such as verifying whether a task could be automated or not. On the other hand, they state that a chatbot would not be of any help to determine the validity of quantitative assessments, such as the optimality of a process. The third one, named *process redesign*, consists in applying existing redesigning techniques in order to improve the quality of the process. Providing such redesign techniques is a suitable task for a chatbot. The fourth and last one, named *process implementation* or *process monitoring* is left as future work as it would require additional tools in addition to the chatbot. They thus evaluated existing LLMs (GPT-1, GPT-2, GPT-3 and GPT-3.5) on their ability to perform such conversational modelling tasks, based on three research questions and seven KPIs. Their conclusion shows that such large language models were suitable for performing such tasks, and will be even more in the future years with the raise of more powerful models, as their experiments show better results for the last models compared to the first ones.

In [VBM23], the authors present several opportunities regarding the capabilities of LLMs to tackle some existing challenges appearing at every stage of the lifecycle of business processes. They focus on six stages of the lifecycle of a business process, namely *identification*, *discovery*, *analysis*, *redesign*, *implementation* and *monitoring*. For the identification phase, they give insights on how LLMs could be used to identify the different processes that take place in a company, based on unstructured documentation. For the process discovery phase, they highlight the fact that LLMs are powerful at summarising text, and valuating the most important elements of a text. They also discuss the possibilities of interacting iteratively with the LLMs by using, for instance, chatbots. For the analysis phase, they state that LLMs can help users by identifying some issues that have a tendency to be repeated. They go even further by questioning whether LLMs could be helpful to localise the source of the issue. For the redesign phase, they discuss the possibility that, given sufficiently precise inputs, LLMs could be able to provide insightful structural modifications of the process in order to mitigate incorrect behaviours, or optimise the process with regards to a given task. For the implementation phase, they describe how LLMs could be used to generate textual explanations of business processes, or how they could behave as valuable chatbots or even process orchestrators by calling APIs and ordering tasks. For the monitoring phase, they present an idea of using LLMs as dashboards chatbots, in order to

give the possibility to the user to observe not a classical, monolithic and possibly partially unsuitable dashboard, but something precisely adapted to her/his needs.

LLM-Based Model Generation

In [KBSvdA24a], the authors present an approach designed to generate automatically process models from their corresponding textual description. They apply several successive (and potentially iterated) steps to generate the process. First, the user has to provide a textual description of the process-to-be. This description is then reworked and enriched with several information to craft the best possible prompt, guiding the LLM towards the generation of a valid, representative model. The prompt thus contains, along with the original description, a role [XYL+25], some knowledge on the expected output structure [MIT23], and some examples of correct [Bro20] and incorrect [MIST24] return values. Although having become classical concepts in prompt engineering, let us detail the additional information provided in the context of this work. The role of the LLM is here to behave as an expert in process modelling, familiar with the classical constructs, and to fill any gap in the provided description. The knowledge given to the LLM is mostly the one of the POWL language [KvZ23], a format introduced by the authors in a previous work. More precisely, the LLM is asked to generate code snippets compliant with functions implemented by the authors with the goal of generating a POWL workflow in the end. The examples provided to the LLM are adding precisions specifying what is a good example (i.e., how the POWL workflow should look like based on an initial description), and what is a bad one (for instance, that a process violating the irreflexivity rule is incorrect). To provide a result, they rely on internal sanity checks to assert that the generated code is compliant with the implementation, and, if not, perform some new exchanges with the LLM until obtaining a correct code snippet. The generated model is then returned to the user who has the possibility to provide feedback aiming at correcting the model, based on an error-handling loop. The authors implemented their approach in the form of the online tool ProMoAI [KBSvdA24b], which was compared to our approach in Chapter 5.

In [EAA+24], the authors present NaLa2BPMN, a tool aiming at generating business processes from natural language descriptions. Their approach is quite similar to ours in terms of expected results. The generation is performed by several successive steps, aiming at retrieving all the components of the process. This is ensured by several successive LLM prompts. These prompts share a common structure consisting of a *role play* field, a *context* field, a *task* field, a *specific instructions* field, an *output format & examples* field, and a *prompt input* field, each providing some context-specific information in order to improve the quality of the LLM's answer. The first exchange with the LLM asks it to reformulate and possibly complete the original user description. According to the authors, this first step has the benefit of enhancing the entities extraction accuracy, which comes in the next steps. From this new version of the description, they ask the LLM to extract activities, along with start and end events, from the description. Once they obtain the names of the activities, they ask the LLM to rephrase the description by replacing the portions of

text describing these activities by their names. Then, they ask the LLM to provide the relationships between the tasks of the process, potentially several times in case of detection of structural inconsistencies. Finally, they get the branching relations from the LLM, in terms of pairs of parallel elements. From all of these information, they perform several algorithmic steps aiming at inserting split and merge gateways, along with eventual loops. The authors implemented their approach in the form of an online tool which was compared to our approach in Chapter 5.

Non-Textual Input Approaches

The authors of [SvdALS23] tackle the problem of automatic generation of business processes from another point of view. Instead of the textual description used as input in all the approaches that we focused on, they take as input a graphical representation of the BPMN process, with the particularity of being hand-drawn. They first present the three specific challenges related to this representation, namely, the shape recognition, the label recognition, and the edge recognition, along with more general challenges raised by this approach. Generically speaking, these challenges are inherently related to the inconsistency of the drawings (e.g., lines may not be straight although they should be), coupled to the similarity of certain elements of the BPMN notation (e.g., start, intermediate, and end events, which differ only by the thickness of their line, or by the number of lines composing them). To solve these problems, the authors first apply a resizing of the image, in order to make it 1,333 pixels long, length required by the Faster R-CNN network [RHGS15] that they use, and which is in charge of detecting the objects contained in the image. Once all the objects have been detected, they perform labels detection to acquire the names of the tasks. To do so, they make use of off-the-shelf optical character recognition (OCR) [SSL03]. To maximise the performances of their approach, along with the quality of the resulting BPMN process, they trained their models on images accompanied by ground-truth annotations describing exactly the elements of the process. Finally, they return the generated process to the user.

The authors of [MBPS24] propose a method for generating business process models from an audio representation of the process. This representation is then converted to text with the help of the Whisper model [RKX+22], which performs speech to text recognition and transformation. Once translated to text, the description must be adjusted in order to comply with an active first-person narrative voice, in which the process steps are explained in a sequential chronological order. It must also conform to some grammar rules defined by the authors. The transformation step may require human interventions to control and potentially alter the generated process, to make it compliant with the needs. The natural language processing step is performed by the spaCy model [HM17], which is useful for its capability of understanding the german language, which is the language adopted in this approach. The experiments, conducted on two processes, show that their approach is able to detect the same number of BPMN elements (activities, flows, events) than in the manually drawn process. However, no information is given on the similarity of the process

	Used Technique	Supported Constructs				Tool Availability	Structured Input	Semantics Preservation	Number of Experiments
		✘	✚	Loops	Unbalancing				
[FMP11, SV17]	NLP, Stanford Parser, Wordnet	✓	✓	✗	✗	✗	✓	?	10
[HKW18]	NLP, SVO Detection, Spreadsheet-Based	✓	✓	✗	✗	✗	✓	?	11
[ISP20]	DSL, Process Mining	✓	✓	✓	✓	✓	✓	?	30
[FSZ21]	Partial Orders, Classical Algorithmic	✓	✓	✗	✗	✓	✓	?	1
[KBSvdA24a]	LLM, POWL	✓	✓	✓	✗	✓	✗	✗	2
[EAA+24]	LLM, Refinement Steps	✓	✓	✓	✗	✓	✗	✗	8
Our approach	LLM, Refinement Steps	✓	✓	✓	✓	✓	✗	✓	~ 200

Table 6.1: Comparison of Similar Modelling Approaches

in terms of syntax or semantics.

Summary and Discussion

The approaches that emerged before 2020 opened the door and paved the way to the modelling and generation of business processes from natural language. However, the advent of LLMs in the 2020's have made almost all these techniques obsolete. Indeed, most of them were using old-fashion natural language processing techniques, thus adding many constraints to the original description which had, for instance, to be compliant with some dictionaries. Then, the pioneer works based on LLMs showed that these tools could be powerfully used to tackle this challenge. Although they remained quite preliminary, they provided several precious advice and illustrated numerous actual challenges that had to be faced in order to take advantage of LLMs. Finally, the most recent works presented several ways of solving this problem, and even transversal approaches based on a different input. The approaches comparable to the one presented in this thesis were reported and succinctly compared in Table 6.1. For each approach, the table presents the technique that is used (Column 1), the supported constructs (Columns 2, 3, 4, and 5), the availability of the tool (Column 6), the requirements regarding the structure of the input (Column 7), the positioning of the approach with regards to semantics preservation (Column 8), and the number of experiments performed (Column 9).

Overall, we showed in the experimental chapter of this thesis that our approach outperformed the (available) existing ones both in terms of accuracy of the generated process, and execution time of the tool.

6.1.2 Generation of LTL

In [FC23], the authors present NL2LTL, a Python package aiming at translating natural language to linear temporal logic. Their approach converts text written in natural language into LTL by making use of patterns in the form of a list that can be enriched by the user beforehand. The natural language processing engine (either Rasa NLU [BFPN17] or GPT) is then prompted, and returns several possible properties, which are ranked by confidence. They also provide some filtering of the result to provide the best candidates to the user, because the patterns that they defined may not be independent (i.e., they may overlap).

In [CHM+23], the authors describe a framework for translating unstructured natural language to LTL. The methodology consists in decomposing the natural language input into sub-translations. Each sub-translation corresponds to a fragment of the formula that will be eventually crafted. This set of sub-translations and the formula can be refined by successive prompts until the user is completely satisfied with the generated formula.

Both of these approaches, although tackling the problem of generating LTL properties from natural language description from different angles (patterns VS successive prompts), were not suitable in the context of this thesis for two main reasons. First, they both require human intervention to refine or select the property to generate. This implies an important knowledge of LTL, which is an assumption that we could not make in this thesis. Second, they both state that the generated property is, in some cases, incorrect with regards to the input description (i.e., it does not precisely exhibit the specified behaviour). In the context of our approach, we must ensure the correctness of the property with regards to its textual description. This is mainly why we decided to focus on a fine-tuning based approach mixed with patterns, as we thought it was giving a good trade-off between verification possibilities and accuracy.

6.2 Refactoring

Despite being frequently found in the literature, the term *refactoring* is evasive and may have several significations. In the context of business processes, the literature shows that it tends to significate "any modification of a business process that eventually optimises it". However, the notion of optimisation is wide and uncertain. For this reason, we will start this section by presenting works that considered refactoring as qualitative improvements of the process. Then, we will present some works that aim at optimising business processes, but which were approached from a different angle. Finally, we will present the work that is the closest to the approaches detailed in this thesis, and which served as thinking basis.

Syntactic Refactoring

The authors of [SM07] present six common mistakes made by developers when modelling with BPMN. Each of them is subject to a refactoring case detailing both the bad practice(s) and the one(s) that should be applied in replacement, called best one(s). The first

refactoring case concerns inconsistent naming of the tasks. The authors state that activity names should not be noun-based, but verb-based and domain specific, should not contain conjunctions such as "and" and "or", and should be short. For gateways, they recommend to leave the name field empty. The second refactoring case stipulates that BPMN processes should not exceed a certain size, and that up to a certain threshold, they should be divided in sub-processes providing the possibility of decreasing the number of visible elements. The third case focuses on gateways, and advices, for instance, to always use them for branching/merging, in order to explicit the control-flow. The fourth case deals with inconsistent use of events, stating that events should not be repeated. Case 5 details how loops should be marked and extracted from the process as unique entities, in order to mitigate the risk of incomprehension. Finally, case 6 describes some common mistakes made when designing graphically a BPMN process, such as non-straight or non-centered flows, or inconstant object sizes.

The authors of [DGKV11] propose a technique for detecting refactoring opportunities in process model repositories. They present four refactoring opportunities. The first one corresponds to a situation where two activities are considered to be the same, but have different labels. This observation is made by a human, and should lead to a renaming of the activities. The second refactoring opportunity consists in analysing *fragments* of the process and assess their similarity. If two fragments are identical, they should be extracted and replaced by a single subprocess. The third opportunity is also linked to this notion of fragment, but concerns fragments partially composed of identical activities and non-similar activities. In such a case, they propose to gather the fragments and specify that the non-similar activities should be skippable. The last opportunity concerns fragments of the process containing activities said "mildly similar", that are, activities with different names and business object of application, although having a similar goal. Similarly, in this case, a subprocess should be created, containing all the activities. To identify such similarities in the process, they transform it into a structure called Refined Process Structure Tree (RPST) [VVK08], decomposing the original process in the so-called fragments. They then define three metrics computable on this structure aiming at quantifying the similarity of each two fragments of the process. Based on these values, they apply their refactoring techniques.

IBUPROFEN [FRPCP13] is a graph-based refactoring approach. It consists of ten algorithms grouped in three categories: *maximisation of relevant elements*, *fine-grained granularity reduction*, and *completeness*. The first algorithm of the first group basically transforms the graph version of the BPMN process into a connected graph by removing all the isolated nodes that it may contain. The second algorithm of this group aims at removing what the authors called *sheet nodes*, which are gateways or intermediate events having no successors. The third algorithm of this group merges the connected gateways of same type into a single gateway. Algorithm 4 basically removes unnecessary flows connecting already connected tasks, while algorithm 5 removes unnecessary gateways, i.e., split gateways with only one outgoing flow. Considering the second group, algorithm 1 creates *compound tasks*, which are replacing tasks followed by several subsequent tasks connected by a round-trip

sequence flow. Algorithm 2 is in charge of combining data objects that are used by a unique task into a single object. Finally, for the last group, algorithm 1 inserts start and end events respectively before and after the starting tasks of the process. The second algorithm of the last group basically inserts gateways between tasks when a task either has several predecessors or several ancestors. To conclude, algorithm 3 refines the names of the activities. The authors also evaluated their tool in an industrial context, and published their results in [PCFRP19].

Non-Fixed Resources Approaches

Works such as [DRS19, DRS21] tackle the optimisation of business processes from another angle. As we have seen in this thesis, a possible way of reducing the execution time of a process consists in adjusting the number of available replicas of the resources needed by the process to execute. In these two works, the business process is encoded in the Maude rewriting logic [CDE$^+$07]. In [DRS19], the authors focus on a static analysis of several metrics computed by executing the process with a given workload. Based on these metrics, they apply different existing optimisation algorithms to compute the optimal pool of resources under the given circumstances. In [DRS21], the approach mostly differs on the way the analysis is performed. Indeed, it is no longer a static analysis performed after the run of the process, but a dynamic analysis aiming at modifying the number of available resources accessible by the process at runtime. This analysis is based on four different strategies. The first one, called *usage-based strategy*, and performed periodically, takes into account the recent usage of a resource to decide on whether to allocate new instances of this resource or not. The second one, called *queue-based strategy*, similar to the previous one, also verifies whether the average number of pending requests of the given resource is inside some defined bounds. If not, some more replicas of the resource are allocated. The third one, called *prediction-based strategy*, makes use of a copy of the process being executed, that is executed in advance for a windowed period of time following the one being analysed. This analysis helps detecting which tasks will be executed, and, consequently, which and how many resources will be required. These metrics are then used to adapt the current pool of resources. The fourth and last strategy, called *combined strategy*, basically combines traits from the three previously detailed strategies, and decides to allocate more resources based on a consensus between them.

The approach proposed in [FSZ24] is rather similar to the one presented in [DRS21]. However, the authors incorporated a predictive analysis of the number of required resources in order to minimise the possibility of resources shortages by predicting in advance the need of each resource for a given incoming period of time. To achieve this, they gathered the resource usage of numerous processes previously simulated with a meaningful workload. They then trained a deep learning model on these information, in order to make it capable of predicting the future number of required resources a short time before the execution, allowing to adapt the pool of resources before reaching the shortage.

Quantitative Refactoring

The authors of [KL22] present an approach for optimising the redesign of process models. This approach picks six out of ten improvement strategies originally introduced in [RM05] and provides a concrete application method for them. The approach first models the process as a set of structural balance equations mimicking the structure of the process (exclusive/parallel gateways), and temporal constraints, representing the tasks executed over time. The first strategy basically makes a task optional so that it can be skipped at runtime, by setting its duration to 0. Strategy 2 can replace a task by an equivalent one of shorter duration to reduce time. Strategy 3 is able to merge two tasks into a single one to benefit from its atomicity. Conversely, strategy 4 may split a task into multiple ones, and put them either in parallel or inside a choice, with the goal of shortening the duration of the process. Similarly, strategy 5 can select two tasks and rearrange them so that they end up in parallel. Finally, strategy 6 is in charge of postponing activities so as to make them executed later, if it can fluidify the execution of the process. Based on these strategies and on the tasks of the process, they build a redesign matrix where the strategies, the tasks, and several metrics associated to each such couple (cost, benefit, etc.) are listed. Finally, they compute several best refactoring possibilities based on the computation of a solver such as CPLEX [Cor13].

In [DS22], the authors introduce the concept of refactoring a business process with the goal of optimising it, and, more precisely, its execution time. To do so, they rely on refactoring operations defined as patterns applied to a given task to make it move somewhere else in the process. These patterns perform in the local environment of the task, that is, its immediate surrounding tasks, gateways, and flows. The first pattern puts a task in parallel of the task preceding it, under the condition that these two tasks do not require the same resource. The second pattern allows a task that follows a merge gateway to enter this gateway. If the gateway is a parallel one, the task is put in parallel of all the tasks that do not require the same resources to execute, and in sequence after the ones with whom it shares resources. If the gateway is an exclusive one, the task is inserted at the end of each path of the gateway, so as to preserve its original non-conditionality. The third pattern is similar to the second one, except that it applies on split gateways instead of merges. The authors also define a notion of *strong flows* representing flows of the process that should be preserved by the refactoring operations. Such flows basically model causal dependencies that should remain so as to preserve the semantics of the process. To reduce the time taken by the exploration of the possible refactored processes, they provide a heuristic driving the selection of the tasks to move and their movement.

Summary and Discussion

The approaches called “refactoring” that can be found in the literature are mostly driven by motivations regarding the improvement of a business process in syntactical directions [SM07, DGKV11, FRPCP13, PCFRP19]. As they provide qualitative analyses of the process, they can not be properly compared to our approach.

The other optimisation methods, such as modifying the number of resources replicas made available to the process [DRS19, DRS21, FSZ24], usually require flexibility in the budget of the companies, as one may have to hire a new employee, or buy a new machine to fulfill the need. Moreover, the only feature on which optimisation is possible is the number of resources. In comparison, our method does not require any budget adjustment (usually an increase), and provides more flexibility in terms of optimisation (one can change the numbers of resources, or keep the same amount of resources but improve the way they are used within the process).

Finally, the quantitative refactoring approaches mentioned lastly are closer to our proposal. In the first one [KL22], some modifications of the process could not be supported by our approach. For instance, by splitting a task into several new tasks, there would be no proper way of ensuring the preservation of the dependencies of the process, nor of its semantics. In the second approach [DS22], processes are executed only once (single instance), and only one replica of each resource is available. These assumptions simplify part of the reasoning and allow the definition of patterns such as the ones presented in the paper, where a task cannot be put in parallel of another if they both require the same resource.

The approaches that we proposed in this thesis handle several instances of the business process being run at the same time, along with multiple replicas of the available resources. Moreover, they preserve the trace semantics of the process, as demonstrated in Chapter 4.

Chapter 7

Conclusion & Perspectives

In this thesis, we tackled two major challenges of the business process management field: the modelling of the processes, and their optimisation via structural refactoring. For these two research questions, we proposed several approaches aiming at simplifying the modelling step, and optimising the business processes of the companies. On the modelling part, the approach that we developed consists in automatically converting a textual description into its corresponding BPMN process, while preserving the constraints that it contains, described in natural language. To perform this generation, the tasks that should appear in the process are extracted from the textual requirements, along with their ordering constraints. Then, a graph comprising all the sequential constraints of the process is built. This graph is converted to an equivalent BPMN representation, and enriched step by step with all the remaining constraints. The resulting BPMN process is then eventually returned to the user. We implemented this approach in the form of a tool that we used for tests and evaluation, with the help of both experts of the BPMN community and novice users.

One step forward in the direction of ensuring the correctness of the generated business process consisted in providing an extension of that tool allowing one to verify some behavioural properties based on their natural language description. This extension of the original work, called GIVUP, showed very good results both in terms of accuracy and performance.

We also proposed three different techniques to optimise business processes enriched with time and resources, all based on refactoring. Although being refactoring-based, these approaches differ in the way they modify the structure of the process. The first one performs a global refactoring of the process, and verifies whether this new version is optimal or not. If not, it applies successive steps backward in order to avoid latencies, while preserving some optimisation. The second and the third approaches address the problem the other way around, by performing several successive small refactoring steps, guiding the process towards an optimised version by following several possible criteria. All of these approaches were implemented and tested on numerous examples. Their evaluation showed satisfactory results both in terms of optimisation and computation time.

7.1 Perspectives

7.1.1 Modelling

The work that we proposed on modelling of business processes offers several axes of improvement.

On the short-term, an idea could be to cross-check the expressions returned by GPT with several other "witness LLMs", which could for instance rate the expressions with regards to the original description. By doing so, some expressions could be kept, some others could be modified, and some others removed. The goal of such an approach would be to reduce the number of erroneous constraints in the generated expressions, consequently leading to more accurate generated processes, and thus better results. Another thing that would be doable in a reasonable time would be to increase the size of our training dataset and moving forward to newer versions of GPT. This would be likely to significantly improve the quality of the generated process.

By enlarging the time window, we could also try to find solutions to add more information or data during the generation phase. For instance, it could be interesting to be able to extract resources and durations from the description, so as to reduce the gap between the generated process and a process that is suitable for applying refactoring operations. Regarding the quality of the process itself, it could be interesting to involve more the user, through exchanges regarding the position of the tasks, or feedback loops to update the process or the expressions corresponding to it.

By considering even further times, it could be interesting to be able to give advices to the modeller at design time, in order to improve the quality of the process. Such advices could be based, for instance, on the best practices in terms of modelling, in order to obtain the most adequate and understandable version of the process. Another interesting thing, rather complex in the context of our approach, would be to enlarge the supported BPMN syntax to, for instance, be able to deal with inclusive gateways, sub-processes, or even to generate more complex workflows such as collaboration diagrams.

Finally, if we widen a bit our field of vision, we could think about more transversal techniques, such as techniques to synchronise the changes made to the generated model and its (obsolete) description.

7.1.2 Verification

Regarding our work on behaviour verification, there is one main perspective that stands out: increasing the number and the complexity of the supported patterns. Indeed, our current fine-tuned GPT model recognises only nine rather simple patterns. This is only a few, among all the possible behaviours that can be described using temporal logics. Thus, finding new widely used patterns would enhance our tool and make it more suitable for real-world verifications.

7.1.3 Refactoring

Despite the promising results offered by our refactoring approaches, there are several areas for improvement and future research.

One thing, that could be thought on the short-term, could be to explore other optimisation techniques for cases where refactoring can not be applied. For instance, if the structure of the process cannot be changed, and if the pool of resources is fixed, it could be interesting to try to schedule the tasks of the process differently at runtime, according to different strategies, in order to fluidify the execution of that process.

In a longer term, another simple—yet complex to apply—idea would be to find better heuristics in order to both shorten the time taken by the approach to complete, and improve the quality of the refactored process. Additionally, the exploration of more advanced optimisation algorithms, such as hybrid metaheuristics, or reinforcement learning techniques, could further improve the efficiency and effectiveness of the process refactoring. These algorithms could be tailored to better navigate the solution space and handle the trade-offs between conflicting objectives. To make the model more realistic, another option could be to enlarge the supported BPMN syntax, or to improve the model of resources to be able to handle, for instance, resources that are not used all the time, or resources that can be produced or consumed.

Two of the main limitations of the current approach are its usage of sequence graphs as underlying structure, and its reliance on simulation to evaluate process candidates. While sequence graphs inherently facilitate the verification and the preservation of the semantics of the process, they also force the original process to be balanced, which is a strong assumption in the real-world. Thus, trying to get rid of the sequence graphs while still guaranteeing the preservation of the semantics of the process is of prime interest. Similarly, although providing accurate performance metrics, simulation can be computationally expensive, especially for large and complex processes. Future work could explore alternative evaluation methods, such as analytical models, or machine learning-based predictors, to reduce computation time while maintaining accuracy. For example, the authors of [ALP23] propose a method to estimate the worst-case execution time of a process model using an SMT solver. Even though their method works for a single instance, and cannot handle loops, it could be a good starting point to develop a more general approach that could be used in our context.

By going a bit out of the scope of the proposed approach, another potential direction for future research is the extension of the approach to handle dynamic and adaptive processes. Real-world business processes often operate in environments where conditions and requirements change over time. Incorporating mechanisms to adapt the refactoring strategy in response to such changes could significantly enhance the applicability of the approach.

Bibliography

[AERDM16] C. Arevalo, M.J. Escalona, I. Ramos, and M. Domínguez-Muñoz. A metamodel to integrate business processes time perspective in BPMN 2.0. *Journal on Information and Software Technology*, 77:17–33, 2016.

[AGMW08] Ahmed Awad, Alexander Grosskopf, Andreas Meyer, and Mathias Weske. Enabling Resource Assignment Constraints in BPMN. 2008.

[AGU72] Alfred Vaino Aho, Michael Randolph Garey, and Jeffrey David Ullman. The Transitive Reduction of a Directed Graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[AHCN24] Somya Agrawal, Thao-Trang Huynh-Cam, and Venkateswarlu Nalluri. Analyzing undergraduate students' perceptions of using chatgpt in daily life. In *Proceedings of the 6th International Workshop on Artificial Intelligence and Education (WAIE'24)*, pages 76–81, 2024.

[ALP23] Muhammad Rizwan Ali, Yngve Lamo, and Violet Ka I Pun. Cost analysis for a resource sensitive workflow modelling language. *Science of Computer Programming*, 225:102896, 2023.

[Ark23] Konstantine Arkoudas. GPT-4 Can't Reason, 2023.

[Bäc96] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, 02 1996.

[BDG22] Patrizio Bellan, Mauro Dragoni, and Chiara Ghidini. Extracting Business Process Entities and Relations from Text Using Pre-trained Language Models and In-Context Learning. In *Proceedings of the 26th International Conference on Enterprise Design, Operations and Computing (EDOC'22)*, 2022.

[BDGP16] Paolo Bocciarelli, Andrea D'Ambrogio, Andrea Giglio, and Emiliano Paglia. In *Proceedings of the Conferenza INCOSE Italia su Systems Engineering (CIISE'16)*, page A BPMN Extension to Enable the Explicit Modeling of Task Resources, 11 2016.

[BFPN17] Tom Bocklisch, Joey Faulker, Nick Pawlowski, and Alan Nichol. Rasa: Open Source Language Understanding and Dialogue Management. 12 2017.

[BGT20] Mihal Brumbulli, Emmanuel Gaudin, and Ciprian Teodorov. Automatic Verification of BPMN Models. In *Proceedings of the 10th European Congress on Embedded Real Time Software and Systems (ERTS'20)*, Toulouse, France, January 2020.

[BJR17] Grady Booch, Ivar Jacobson, and James Rumbaugh. Unified Modelling Language (UML), Version 2.5.1, 2017.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[BKO10] Dominik Q. Birkmeier, Sebastian Klöckner, and Sven Overhage. An Empirical Comparison of the Usability of BPMN and UML Activity Diagrams for Business Users. In *Proceedings of the 18th European Conference on Information Systems (ECIS'10)*, ECIS'10, 2010.

[BLS17] Gianluca Barbon, Vincent Leroy, and Gwen Salaün. Debugging of Concurrent Systems Using Counterexample Analysis. In *Proceedings of the 7th International Conference on Fundamentals of Software Engineering (FSEN'17)*, LNCS, pages 20–34. Springer, 2017.

[BNE07] Nicola Beume, Boris Naujoks, and Michael Emmerich. SMS-EMOA: Multiobjective selection based on dominated hypervolume. *European Journal of Operational Research*, 181(3):1653–1669, 2007.

[BOQ+25] Marialena Bevilacqua, Kezia Oketch, Ruiyang Qin, Will Stamey, Xinyuan Zhang, Yi Gan, Kai Yang, and Ahmed Abbasi. When Automated Assessment Meets Automated Content Generation: Examining Text Quality in the Era of GPTs. *ACM Transactions on Information Systems*, 43(2), jan 2025.

[BRJ05] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[Bro20] Tom B. et al. Brown. Language Models are Few-Shot Learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20)*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[BSS15] Marlon Alexander Braun, Pradyumn Kumar Shukla, and Hartmut Schmeck. Obtaining Optimal Pareto Front Approximations using Scalarized Preference Information. In *Proceedings of the 17th Annual Conference on Genetic and Evolutionary Computation (GECCO'15)*, page 631–638, New York, NY, USA, 2015. Association for Computing Machinery.

[Büc66] Julius Richard Büchi. Symposium on Decision Problems: On a Decision Method in Restricted Second Order Arithmetic. In Ernest Nagel, Patrick Suppes, and Alfred Tarski, editors, *Logic, Methodology and Philosophy of Science*, volume 44 of *Studies in Logic and the Foundations of Mathematics*, pages 1–11. Elsevier, 1966.

[BvdAD+22] Patrizio Bellan, Han van der Aa, Mauro Dragoni, Chiara Ghidini, and Simone Paolo Ponzetto. PET: An Annotated Dataset for Process Extraction from Natural Language Text Tasks. In *Business Process Management Workshops*, volume 460 of *LNBIP*, pages 315–321. Springer, 2022.

[BW10] Edward Anton Bender and Stanley Gill Williamson. *Lists, Decisions and Graphs - With an Introduction to Probability*. University of California, San Diego, 2010.

[BZOW19] Ekaterina Bazhenova, Francesca Zerbato, Barbara Oliboni, and Mathias Weske. From BPMN process models to DMN decision models. *Information Systems*, 83:69–88, 2019.

[BZS18] Alexandru Burdusel, Steffen Zschaler, and Daniel Strüber. MDEoptimiser: A Search Based Model Engineering Tool. In *Proceedings of the 21st International Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 12–16. ACM, 2018.

[CCG+11] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Christine McKinty, Vincent Powazny, Frédéric Lang, Wendelin Serwe, and Gideon Smeding. Reference Manual of the LNT to LOTOS Translator (Version 7.5). INRIA/VASY, 2011.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude – A High-Performance Logical Framework*. Springer-Verlag, Berlin, Heidelberg, 2007.

[CDGBR16] Raffaele Conforti, Marlon Dumas, Luciano García-Bañuelos, and Marcello La Rosa. BPMN Miner: Automated Discovery of BPMN Process Models with Hierarchical Structure. *Information Systems*, 56:284–303, 2016.

[CDNS25] David Cremer, Benjamin Dalmas, Quentin Nivon, and Gwen Salaün. Incremental Synchronization of BPMN Models and Documentations by Leveraging Structural Algorithms and LLMs. In *Proceedings of the 31st International Conference on Cooperative Information Systems (CoopIS'25)*, pages 1–18, Marbella, Spain, October 2025.

[CE82] Edmund Melson Clarke and Ernest Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[CFSZ22] Angel Contreras, Yliès Falcone, Gwen Salaün, and Ahang Zuo. WEASY: A Tool for Modelling Optimised BPMN Processes. In *Proceedings of the 18th International Conference on Formal Aspects of Component Software (FACS'22)*, Oslo / Online, Norway, Nov 2022.

[Che76] Peter Pin-Shan Chen. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, mar 1976.

[CHM+23] Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models. In *Proceedings of the 35th International Conference on Computer Aided Verification (CAV'23)*, volume 13965 of *LNCS*, pages 383–396. Springer, 2023.

[Chu24] Kenneth Church. Emerging trends: When can users trust GPT, and when should they intervene? *Natural Language Engineering*, 30(2):417–427, 2024.

[CMP+20] Flavio Corradini, Andrea Morichetta, Andrea Polini, Barbara Re, Lorenzo Rossi, and Francesco Tiezzi. Correctness checking for BPMN collaborations with sub-processes. *Journal of Systems and Software*, 166:110594, 2020.

[Cor13] IBM Corporation. IBM ILOG CPLEX 12.6 User Manual, 2013.

[CW71] Harold Chestnut and Stanley Williams. Business process modeling improves administrative control. In *The Impact of Information Technology on Management Operation*, 1971.

[dABTS+18] Ana Cláudia de Almeida Bordignon, Lucinéia Heloisa Thom, Thanner Soares Silva, Vinicius Stein Dani, Marcelo Fantinato, and Renato Cesar Borges Ferreira. Natural Language Processing in Business Process Identification and Modeling: A Systematic Literature Review. In *Proceedings of the 14th Brazilian Symposium on Information Systems (SBSI'18)*, SBSI '18, New York, NY, USA, 2018. Association for Computing Machinery.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James Curtis Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'99)*, pages 411–420, 1999.

[Dav93] Thomas Hayes Davenport. *Process innovation: reengineering work through information technology*. Harvard Business School Press, USA, 1993.

[DDO08] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Journal of Information and Software Technology*, 50(12):1281–1294, 2008.

[DFR+22] Francisco Durán, Yliès Falcone, Camilo Rocha, Gwen Salaün, and Ahang Zuo. From Static to Dynamic Analysis and Allocation of Resources for BPMN Processes. In *Proceedings of the 14th International Workshop on Rewriting Logic and its Applications (WRLA'22)*, pages 1–18. Springer, 2022.

[DGKV11] Remco M. Dijkman, Beat Gfeller, Jochen Malte Küster, and Hagen Völzer. Identifying Refactoring Opportunities in Process Model Repositories. *Journal of Information and Software Technology*, 53(9):937–948, 2011.

[DLRC+22] Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, and Henrich Lauko. From Spot 2.0 to Spot 2.10: What's New? In *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22)*, pages 174–187. Springer International Publishing, 2022.

[dMMM06] Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. Generating Typed Dependency Parses from Phrase Structure Parses. In Nicoletta Calzolari, Khalid Choukri, Aldo Gangemi, Bente Maegaard, Joseph Mariani, Jan Odijk, and Daniel Tapias, editors, *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC'06)*, Genoa, Italy, may 2006. European Language Resources Association (ELRA).

[DN11] Juan José Durillo and Antonio Jesus Nebro. jMetal: A Java Framework for Multi-objective Optimization. *Journal of Advances in Engineering Software*, 42(10):760–771, 2011.

[DPAM02] Kalyanmoy Deb, Amrit Pratap, Sapan Agarwal, and Thirunavukaras Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[DRS18a] Francisco Durán, Camilo Rocha, and Gwen Salaün. Computing the Parallelism Degree of Timed BPMN Processes. In *Proceedings of the 16th International Workshop on Foundations of Coordination Languages and Self-Adaptative Systems (FOCLASA'18)*, pages 1–16, 2018.

[DRS18b] Francisco Durán, Camilo Rocha, and Gwen Salaün. Stochastic Analysis of BPMN with Time in Rewriting Logic. *Science of Computer Programming*, 168:1–17, 2018.

[DRS18c] Francisco Durán, Camilo Rocha, and Gwen Salaün. Symbolic Specification and Verification of Data-Aware BPMN Processes Using Rewriting Modulo SMT. In Vlad Rusu, editor, *Proceedings of the 12th International Workshop on Rewriting Logic and its Applications (WRLA'18)*, volume 11152 of *LNCS*, pages 76–97. Springer, 2018.

[DRS19] Francisco Durán, Camilo Rocha, and Gwen Salaün. A Rewriting Logic Approach to Resource Allocation Analysis in Business Process Models. *Science of Computer Programming*, 183, 2019.

[DRS21] Francisco Durán, Camilo Rocha, and Gwen Salaün. Resource provisioning strategies for BPMN processes: Specification and analysis using maude. *Journal of Logical and Algebraic Methods in Programming*, 123:100711, 2021.

[DS22] Francisco Durán and Gwen Salaün. Optimization of BPMN Processes via Automated Refactoring. In *Proceedings of the 20th International Conference on Service-Oriented Computing (ICSOC'22)*, volume 13740 of *LNCS*, pages 3–18. Springer, 2022.

[ea24a] Gemini Team et al. Gemini: A Family of Highly Capable Multimodal Models, 2024.

[ea24b] OpenAI et al. GPT-4 Technical Report, 2024.

[EAA+24] Ali Nour Eldin, Nour Assy, Olan Anesini, Benjamin Dalmas, and Walid Gaaloul. A Decomposed Hybrid Approach to Business Process Modeling with LLMs. In *Proceedings of the 30th International Conference on Cooperative Information Systems (CoopIS'24)*, 2024.

[EG16] Rik Eshuis and Pieter Van Gorp. Synthesizing data-centric models from business process models. *Computing*, 98(4):345–373, 2016.

[EH83] Eernest Allen Emerson and Joseph Yehuda Halpern. "sometimes" and "not never" revisited: on branching versus linear time (preliminary report). In *Proceedings of the 10th Symposium on Principles of Programming Languages (POPL'83)*, page 127–140, New York, NY, USA, 1983. Association for Computing Machinery.

[FC23] Francesco Fuggitti and Tathagata Chakraborti. NL2LTL – a Python Package for Converting Natural Language (NL) Instructions to Linear Temporal Logic (LTL) Formulas. In *Proceedings of the 37th International Conference on Artificial Intelligence (AAAI'23)*, pages 16428–16430. AAAI Press, 2023.

[FFK23] Hans-Georg Fill, Peter Fettke, and Julius Köpke. Conceptual Modeling and Large Language Models: Impressions From First Experiments With ChatGPT. pages 1–15, 2023.

[FMP11] Fabian Friedrich, Jan Mendling, and Frank Puhlmann. Process Model Generation from Natural Language Text. In *Proceedings of the 23rd International Conference on Advanced Information Systems Engineering (CAiSE'11)*, pages 482–496, 06 2011.

[FNS24] Irman Faqrizal, Quentin Nivon, and Gwen Salaün. Automated Repair of Violated Eventually Properties in Concurrent Programs. In *Proceedings of the 12th International Conference on Formal Methods in Software Engineering (FormaliSE'24)*, pages 1–11, Lisbon, Portugal, April 2024. IEEE.

[FR21] Peter Fettke and Wolfgang Reisig. Handbook of heraklit, 2021.

[FRPCP13] María Fernández-Ropero, Ricardo Pérez-Castillo, and Mario Piattini. Graph-Based Business Process Model Refactoring. In *Proceedings of the 3rd International Symposium on Data-driven Process Discovery and Analysis (SIMPDA'13)*, volume 1027 of *CEUR Workshop Proceedings*, pages 16–30, 2013.

[FS23] Radha Firaina and Dwi Sulisworo. Exploring the usage of chatgpt in higher education: Frequency and impact on productivity. *Buletin Edukasi Indonesia*, 2:39–46, 03 2023.

[FSZ21] Yliès Falcone, Gwen Salaün, and Ahang Zuo. Semi-automated Modelling of Optimized BPMN Processes. In *Proceedings of the 18th International Conference on Services Computing (SCC'21)*, pages 425–430. IEEE, 2021.

[FSZ22] Yliès Falcone, Gwen Salaün, and Ahang Zuo. Probabilistic Model Checking of BPMN Processes at Runtime. In *Proceedings of the 17th International Conference on integrated Formal Methods (iFM'22)*, pages 1–17. Springer International Publishing, 2022.

[FSZ24] Yliès Falcone, Gwen Salaün, and Ahang Zuo. Dynamic Resource Allocation for Executable BPMN Processes Leveraging Predictive Analytics. In *Proceedings of the 24th International Conference on Software Quality, Reliability and Security (QRS'24)*, pages 689–700, 2024.

[Gan10] Henry Laurence Gantt. *Work, Wages and Profits*. 1910.

[Gea12] Cristina Venera Geambasu. BPMN vs. UML Activity Diagram for Business Process Modeling. *Journal of Accounting and Management Information Systems*, 11(4):637–651, 2012.

[GG21] Frank Bunker Gilbreth and Lillian Moller Gilbreth. Process Charts: First Steps in Finding the One Best Way to Do Work. *Transactions of the American Society of Mechanical Engineers*, 43:1029–1043, 12 1921.

[GJ07] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Monographs in Computer Science. Springer New York, NY, 2007.

[GL01] Hubert Garavel and Frédéric Lang. SVL: A Scripting Language for Compositional Verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proceedings of the 21st International Conference on Formal Techniques (FORTE'01)*, pages 377–392, Boston, MA, 2001. Springer US.

[Gro07] Alexander Großkopf. An Extended Resource Information Layer for BPMN. 01 2007.

[Gro22] Norbert Gronau. Errors in the Process of Modeling Business Processes. In Boris Shishkov, editor, *Business Modeling and Software Design*, pages 221–229, Cham, 2022. Springer International Publishing.

[GT98] Dimitrios Georgakopoulos and Aphrodite Tsalgatidou. *Technology and Tools for Comprehensive Business Process Lifecycle Management*, pages 356–395. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[GT09] Denis Gagne and André Trudel. Time-bpmn. In *Proceedings of the 11th International Conference on E-Commerce Technology Workshops (CEC'09)*, pages 361–367, 2009.

[GtH23] Bert Gordijn and Henk ten Have. ChatGPT: evolution or revolution? In *Medicine, health care, and philosophy*, volume 26, pages 1–2, 2023.

[GV06] Stijn Goedertier and Jan Vanthienen. Designing compliant business processes with obligations and permissions. In Johann Eder and Schahram Dustdar, editors, *Business Process Management Workshops*, pages 5–14, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[HC93] Michael Hammer and James Champy. *Reengineering the Corporation: A Manifesto for Business Revolution*. Harper Collins, 1993.

[HKW18] Krzysztof Honkisz, Krzysztof Kluza, and Piotr Wiśniewski. A Concept for Generating Business Process Models from Natural Language Description. In *Proceedings of the 11th International Conference on Knowledge Science, Engineering and Management (KSEM'18)*, pages 91–103, 2018.

[HM17] Matthew Honnibal and Ines Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. *To appear*, 2017.

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[HS12] Luke T. Herbert and Robin Sharp. Quantitative Analysis of Probabilistic BPMN Workflows. volume Volume 2: 32nd Computers and Information in Engineering Conference, Parts A and B of *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 509–518, 08 2012.

[ide98] IEEE Standard for Functional Modeling Language - Syntax and Semantics for IDEF0. *IEEE Std 1320.1-1998*, pages 1–116, 1998.

[ISP20] Ana Ivanchikj, Souhaila Serbout, and Cesare Pautasso. From Text to Visual BPMN Process Models: Design and Evaluation. In *Proceedings of the 23rd International Conference on Model Driven Engineering Languages and Sys-*

tems (MODELS'20), pages 229–239. Association for Computing Machinery, 2020.

[Jac20] Philip Jackson. Understanding understanding and ambiguity in natural language. *Procedia Computer Science*, 169:209–225, 2020. Postproceedings of the 10th Annual International Conference on Biologically Inspired Cognitive Architectures, BICA 2019 (Tenth Annual Meeting of the BICA Society), held August 15-19, 2019 in Seattle, Washington, USA.

[JMP93] Henry J. Johansson, Patrick McHugh, and A. John Pendlebury. *Business Process Reengineering: BreakPoint Strategies for Market Dominance*. 1993.

[JN06] John Jeston and Johan Nelis. *Business Process Management*. Elsevier, 2006.

[KBdL+18] Anna Kalenkova, Andrea Burattin, Massimiliano de Leoni, Wil van der Aalst, and Alessandro Sperduti. Discovering High-level BPMN Process Models from Event Data. *Business Process Management Journal*, 25(5):995–1019, 10 2018.

[KBK+23] Nataliia Klievtsova, Janik-Vasily Benzin, Timotheus Kampik, Juergen Mangler, and Stefanie Rinderle-Ma. Conversational Process Modelling: State of the Art, Applications, and Implications in Practice. In *Proceedings of the 21st Business Process Management Forum (BPM'23 Forum)*, 2023.

[KBSvdA24a] Humam Kourani, Alessandro Berti, Daniel Schuster, and Willibrordus Martinus Pancratius van der Aalst. Process Modeling with Large Language Models. In *Proceedings of the 25th International Conference on Business Process Modeling, Development and Support (BPMDS'24)*, volume 511 of *LNBIP*, pages 229–244. Springer, 2024.

[KBSvdA24b] Humam Kourani, Alessandro Berti, Daniel Schuster, and Willibrordus Martinus Pancratius van der Aalst. ProMoAI: Process Modeling with Generative AI. In *Proceedings of the 33rd International Joint Conference on Artificial Intelligence (IJCAI'24)*, 2024.

[KC99] Joshua Knowles and David Corne. The Pareto Archived Evolution Strategy: a new Baseline Algorithm for Pareto Multiobjective Optimisation. In *Proceedings of the 6th Congress on Evolutionary Computation (CEC'99)*, volume 1, pages 98–105, 1999.

[KGV83] Scott Kirkpatrick, C. Daniel Gelatt, and Mario Pietro Vecchi. Optimization by Simulated Annealing. *Science (New York, N.Y.)*, 220:671–80, 06 1983.

[KL22] Akhil Kumar and Rong Liu. Business Workflow Optimization through Process Model Redesign. In *IEEE Transactions on Engineering Management*, LNCS, pages 3068–3084. Springer, 2022.

[KO10] Agnes Koschmider and Andreas Oberweis. *Designing Business Processes with a Recommendation-Based Editor*, pages 299–312. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[KPS17] Ajay Krishna, Pascal Poizat, and Gwen Salaün. VBPMN: Automated Verification of BPMN Processes. In *Proceedings of the 13th International Conference on integrated Formal Methods (iFM'17)*, LNCS, pages 323–331. Springer, 2017.

[KPS19] Ajay Krishna, Pascal Poizat, and Gwen Salaün. Checking Business Process Evolution. *Science of Computer Programming*, 170:1–26, 2019.

[KvZ23] Humam Kourani and Sebastiaan J. van Zelst. POWL: Partially Ordered Workflow Language. In Chiara Di Francescomarino, Andrea Burattin, Christian Janiesch, and Shazia Sadiq, editors, *Proceedings of the 21st International Conference on Business Process Management (BPM'23)*, pages 92–108, Cham, 2023. Springer Nature Switzerland.

[Lan05] Frédéric Lang. Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-The-Fly Verification Methods. In Judi Romijn, Graeme Smith, and Jaco van de Pol, editors, *Proceedings of the 5th International Conference on integrated Formal Methods (iFM'05)*, pages 70–88, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[LF14] Niels Lohmann and Dirk Fahland. Where did i go wrong? In Shazia Sadiq, Pnina Soffer, and Hagen Völzer, editors, *Business Process Management*, pages 283–300, Cham, 2014. Springer International Publishing.

[Lin96] Dekang Lin. On the Structural Complexity of Natural Language Sentences. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING'96)*, 1996.

[MBPS24] Madline Maria Mößlang, Reinhard Bernsteiner, Christian Ploder, and Stephan Schlögl. Automatic Generation of a Business Process Model Diagram Based on Natural Language Processing. In *Proceedings of the 18th International Conference on Knowledge Management in Organisations (KMO'24)*, LNBIP. Springer, 2024.

[Mil89] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., USA, 1989.

[Mil95] George Armitage Miller. WordNet: A Lexical Database for English, 1995.

[MIST24] Daiki Miyake, Akihiro Iohara, Yu Saito, and Toshiyuki Tanaka. Negative-prompt Inversion: Fast Image Inversion for Editing with Text-guided Diffusion Models, 2024.

[MIT23] Ariana Martino, Michael Iannelli, and Coleen Truong. Knowledge Injection to Counter Large Language Model (LLM) Hallucination. In Catia Pesquita, Hala Skaf-Molli, Vasilis Efthymiou, Sabrina Kirrane, Axel Ngonga, Diego Collarana, Renato Cerqueira, Mehwish Alam, Cassia Trojahn, and Sven Hertling, editors, *Proceedings of the 20th Extended Semantic Web Conference (ESWC'23)*, pages 182–185, Cham, 2023. Springer Nature Switzerland.

[MKPC14] Rinaldo Morais, Samir Kazan, Silvia Pádua, and André Costa. An analysis of BPM lifecycles: From a literature review to a framework proposal. *Business Process Management Journal*, 20, 05 2014.

[ML04] Manuel Mazzara and Roberto Lucchi. A Framework for Generic Error Handling in Business Processes. *Electronic Notes in Theoretical Computer Science*, 105:133–145, 2004.

[MR08] Michael zur Muehlen and Jan Recker. How much language is enough? theoretical and practical use of the business process modeling notation. In Zohra Bellahsène and Michel Léonard, editors, *Advanced Information Systems Engineering*, pages 465–479, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[MRCF59] Donald G. Malcolm, J.H. Roseboom, C.E. Clark, and W. Fazar. Application of a Technique for Research and Development Program Evaluation. *Operations Research*, 7:646–669, 1959.

[MT08] Radu Mateescu and Damien Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In Jorge Cuellar and Tom Maibaum, editors, *Proceedings of the 15th International Symposium on Formal Methods (FM'08)*, volume 5014, pages 148–164, Turku, Finland, May 2008. Springer Verlag.

[MW08] Derek Miers and Stephen Andrew White. *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. Future Strategies, Inc., 2008.

[Neu09] Thomas Neubauer. An empirical study about the status of business process management. *Business Process Management Journal*, 15:166–183, 04 2009.

[NK06] Anna Gunhild Nysetvold and John Krogstie. Assessing Business Process Modeling Languages Using a Generic Quality Framework. In *Advanced Topics in Database Research*, volume 5, pages 79–93. Keng Siau, 2006.

[NS22] Quentin Nivon and Gwen Salaün. Debugging of BPMN Processes Using Coloring Techniques. In *Proceedings of the 18th International Conference on Formal Aspects of Components Softwares (FACS'22)*, pages 90–109. Springer, 2022.

[NS25] Quentin Nivon and Gwen Salaün. GIVUP: Automated Generation and Verification of Textual Process Descriptions. In *Proceedings of the 33rd In-*

ternational Conference on Foundations of Software Engineering (FSE'25), 2025.

[OLRR12] César Augusto L. Oliveira, Ricardo Massa F. Lima, Hajo A. Reijers, and Joel Tiago S. Ribeiro. Quantitative Analysis of Resource-Constrained Business Processes. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 42(3):669–684, 2012.

[OMG11] OMG. Business Process Model and Notation (BPMN), Version 2.0, January 2011.

[Pan12] Theodore Panagacos. *The Ultimate Guide to Business Process Management: Everything you need to know and how to apply it to your organization.* Amazon Digital Services LLC - KDP Print US, 2012.

[Pan19] Annibale Panichella. An Adaptive Evolutionary Algorithm based on Non-Euclidean Geometry for Many-objective Optimization. In *Proceedings of the 21st Annual Conference on Genetic and Evolutionary Computation (GECCO'19)*, page 595–603, New York, NY, USA, 2019. Association for Computing Machinery.

[Pan22] Annibale Panichella. An Improved Pareto Front Modeling Algorithm for Large-scale Many-Objective Optimization. In *Proceedings of the 24th Annual Conference on Genetic and Evolutionary Computation (GECCO'22)*, page 565–573, New York, NY, USA, 2022. Association for Computing Machinery.

[Par81] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science*, pages 167–183, Berlin, Heidelberg, 1981. Springer Berlin Heidelberg.

[PCFRP19] Ricardo Pérez-Castillo, María Fernández-Ropero, and Mario Piattini. Business process model refactoring applying IBUPROFEN. An industrial evaluation. *Journal of Systems and Software*, 147:86–103, 2019.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS'77)*, pages 46–57, 1977.

[QSX+20] XiPeng Qiu, TianXiang Sun, YiGe Xu, YunFan Shao, Ning Dai, and XuanJing Huang. Pre-trained Models for Natural Language Processing: A Survey. *Science China Technological Sciences*, 63(10):1872–1897, sep 2020.

[RB90] Geary A. Rummler and Alan P. Brache. *Improving Performance: How to Manage the White Space on the Organization Chart.* 1990.

[RH07] Tomislav Rozman and Romana Vajde Horvat. Analysis of most common proces modelling mistakes in BPMN process models. In *Proceedings of the*

14th European Conference on Software Process Improvement (EuroSPI'07), 01 2007.

[RHB15] Manfred Reichert, Alena Hallerbach, and Thomas Bauer. *Lifecycle Management of Business Process Variants*, pages 251–278. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[RHGS15] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In *Proceedings of the 29th International Conference on Neural Information Processing Systems (NeurIPS'15)*, 2015.

[RKX+22] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust Speech Recognition via Large-Scale Weak Supervision, 2022.

[RM05] H.A. Reijers and S. Liman Mansar. Best practices in business process redesign: an overview and qualitative evaluation of successful redesign heuristics. *Omega*, 33(4):283–306, 2005.

[RNSS18] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving Language Understanding by Generative Pre-Training. 2018.

[RSBR14] Suman Roy, A.S.M. Sajeev, Sidharth Bihary, and Abhishek Ranjan. An Empirical Study of Error Patterns in Industrial Business Process Models. *IEEE Transactions on Services Computing*, 7(2):140–153, 2014.

[RvSK14] Mark von Rosing, Henrik von Scheel, and Alex Kokkonen. *The Complete Business Process Handbook: Body of Knowledge from Process Modeling to BPM*. 12 2014.

[Sal22] Gwen Salaün. Quantifying the Similarity of BPMN Processes. In *Proceedings of the 29th Asia-Pacific Software Engineering Conference (APSEC'22)*, pages 1–10, Virtual, Japan, Dec 2022.

[SCV11] Luis Jesús Ramón Stroppi, Omar Chiotti, and Pablo David Villarreal. A BPMN 2.0 Extension to Define the Resource Perspective of Business Process Models. In *Proceedings of the 1st Conferencia Iberoamericana de Software Engineering (CIbSE'11)*, 2011.

[SFS11] G.N. Suarez, J. Freund, and M. Schrepfer. *Best practice guidelines for BPMN 2.0*. Future Strategies, Inc., 2011.

[Shi24] Deliang Shi. Use ChatGPT to maximize everyday efficiency. *Naturalis Scientias*, 01:87–99, 01 2024.

[SM07] Darius Silingas and Edita Mileviciene. *Refactoring BPMN Models: From 'Bad Smells' to Best Practices and Patterns*. 01 2007.

[SMA24] Laith R. Sultan, Mohamed Kh Mohamed, and Savvas Andronikou. ChatGPT-4: a breakthrough in ultrasound image analysis. *Radiology Advances*, 1(1), 03 2024.

[Smi76] Adam Smith. *An Inquiry into the Nature and Causes of the Wealth of Nations*. University Press, 1776.

[SMK23] Kaya Stechly, Matthew Marquez, and Subbarao Kambhampati. GPT-4 Doesn't Know It's Wrong: An Analysis of Iterative Prompting for Reasoning Problems, 2023.

[SSL03] Sargur Narasimhamurthy Srihari, Ajay Shekhawat, and Stephen W. Lam. *Optical Character Recognition (OCR)*, page 1326–1333. John Wiley and Sons Ltd., GBR, 2003.

[Str17] Daniel Strüber. Generating Efficient Mutation Operators for Search-Based Model-Driven Engineering. In *Proceedings of the 10th International Conference on Model Transformation (ICMT'17)*, volume 10374 of *Lecture Notes in Computer Science*, pages 121–137. Springer, 2017.

[SV17] Konstantinos Sintoris and Kostas Vergidis. Extracting Business Process Models Using Natural Language Processing (NLP) Techniques. In *Proceedings of the 19th International Conference on Business Informatics (CBI'17)*, pages 135–139, 2017.

[SvdALS23] Bernhard Schäfer, Han van der Aa, Henrik Leopold, and Heiner Stuckenschmidt. Sketch2Process: End-to-End BPMN Sketch Recognition Based on Neural Networks. *IEEE Transactions on Software Engineering*, 49(4):2621–2641, 2023.

[SWN25] Thomas R. Shultz, Jamie M. Wise, and Ardavan S. Nobandegani. Text Understanding in GPT-4 vs Humans, 2025.

[Sza23] Attila Szabo. ChatGPT is a Breakthrough in Science and Education but Fails a Test in Sports and Exercise Psychology. volume 1, pages 24–40, 05 2023.

[Tay11] Frederick Winslow Taylor. *The Principles of Scientific Management*. 1911.

[Tur50] Alan Mathison Turing. Computing machinery and intelligence. *Mind*, LIX(236):433–460, 10 1950.

[VBM23] Maxim Vidgof, Stefan Bachhofner, and Jan Mendling. Large Language Models for Business Process Management: Opportunities and Challenges. In *Proceedings of the 21st Business Process Management Forum (BPM'23 Forum)*, 2023.

[vdA16] Willibrordus Martinus Pancratius van der Aalst. *Process Mining*. Springer Berlin, Heidelberg, 2016.

[VTS22] Pedro Valderas, Victoriav Torres, and Estefanía Serral. Modelling and executing IoT-enhanced business processes through BPMN and microservices. *Journal of Systems and Software*, 184:111139, 2022.

[VVK08] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. The Refined Process Structure Tree. In Marlon Dumas, Manfred Reichert, and Ming-Chien Shan, editors, *Proceedings of the 6th International Conference on Business Process Management (BPM'08)*, pages 100–115, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[WCF+25] Xiao-Kun Wu, Naipeng Chao, Giancarlo Fortino, Yonglin Tian, Qingguo Meng, Jia Liu, Kaibin Huang, Long Hu, Mohsen Guizani, Wanyi Li, Yanru Pan, Yixue Hao, Rui Wang, Fei-Yue Wang, Dusit Niyato, Kai Hwang, Fei Lin, Min Chen, and Limeng Lu. LLM Fine-Tuning: Concepts, Opportunities, and Challenges. *Big Data and Cognitive Computing*, 9(4):87, April 2025.

[WCP+23] Boxin Wang, Weixin Chen, Hengzhi Pei, Chulin Xie, Mintong Kang, Chenhui Zhang, Chejian Xu, Zidi Xiong, Ritik Dutta, Rylan Schaeffer, Sang T. Truong, Simran Arora, Manias Mazeika, Dan Hendrycks, Zinan Lin, Yu Cheng, Sanmi Koyejo, Dawn Song, and Bo Li. DecodingTrust: A Comprehensive Assessment of Trustworthiness in GPT Models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (NIPS'23)*, NIPS'23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[Wes07] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007.

[WG09] Peter Y.H. Wong and Jeremy Gibbons. A Relative Timed Semantics for BPMN. *Electronic Notes in Theoretical Computer Science*, 229(2):59–75, 2009. Proceedings of the 7th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2008).

[Whi04] Stephen White. Process Modeling Notations and Workflow Patterns. 5, 2004.

[WLW+23] Haifeng Wang, Jiwei Li, Hua Wu, Eduard Hovy, and Yu Sun. Pre-Trained Language Models and Their Applications. *Engineering*, 25:51–65, 2023.

[WM24] Vinzenz Wolf and Christian Maier. ChatGPT usage in everyday life: A motivation-theoretic mixed-methods study. *International Journal of Information Management*, 79:102821, 2024.

[XYL+25] Benfeng Xu, An Yang, Junyang Lin, Quan Wang, Chang Zhou, Yongdong Zhang, and Zhendong Mao. ExpertPrompting: Instructing Large Language Models to be Distinguished Experts, 2025.

[ZLC+24] Min Zhao, Fuan Li, Francis Cai, Haiyang Chen, and Zheng Li. Can we trust LLMs to help us? An examination of the potential use of GPT-4 in generating quality literature reviews. *Nankai Business Review International*, 16, 09 2024.

[ZLT01] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. *TIK-Report*, 103, 07 2001.

[ZZL+25] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A Survey of Large Language Models, 2025.

Appendices

Appendix A

Example of Prompts

A.1 Example of System Prompt

Component	Prompt Content
Agent Role	`You are a helpful assistant which aims at analysing and extracting the relationships that exist between the different tasks of the textual description that is given to you.`
Expected Output Format	`Your answer will be composed of one or several lines starting with the '-' symbol containing each an expression representing one or several relationships between the tasks. The tasks that you discover can be related to each others in five different ways. (1) The tasks can be ordered sequentially, meaning that some of them have to be executed before some others. In such a case, they are separated by the '<' symbol. (2) The tasks can be mutually exclusive, meaning that only one of them can be executed. In such a case, they are separated by the '|' symbol. (3) The tasks can be parallelisable, meaning that they can all execute simultaneously. In such a case, they are separated by the '&' symbol. (4) The tasks can be repeated, meaning that they can be executed several times during the lifecycle of the process. In such a case, they are put between parenthesis in which they are separated by a coma, and the '*' symbol is put next to the closing parenthesis. (5) When no information is given about tasks, when you need to list tasks without giving their relationship, or when you do not know how some tasks are related, you can separate them using the ',' symbol.`
Example of Correct Output	`For instance, from the description 'I want to do A and B in parallel, followed by C or D', you can return the expression '- (A & B) < (C | D)'.`
Advice of Undesired Behaviour	`Lastly, you must not use any other symbol than those listed before.`

The system prompt contains four identifiable parts, related to the four best prompting practices that can be found in the literature. The *agent role* specifies the role that the agent should take [XYL+25], the *expected output format* describes in details how the agent should format its output [MIT23], the *example of correct output* gives the agent an example of correct answer corresponding to a given correct input [Bro20], and the *advice of undesired behaviour* tells the agent what it should not do, or provide as answer [MIST24].

A.2 Example of User Prompt

```
Task A followed by task B in parallel with task C followed by task D.
```

The user prompt is an example of description of a business process that a user may write.

A.3 Example of Assistant Prompt

```
\n- (A < B) & (C < D)
```

The assistant prompt is the answer expected by GPT according to the user prompt. Note that, for practical reasons, this prompt can be decomposed in several lines starting by the character '\n' followed by the character '-'.

A.4 Example of Training/Validation File Content

Role	Prompt Content
System	Full prompt of Appendix A.1.
User	`CollectGoods is followed by PrepareParcel, itself followed by Payment. Then, either DeliveryByCar or DeliveryByDrone execute, but not both.`
Assistant	`- CollectGoods < PrepareParcel < Payment` `- (CollectGoods, PrepareParcel, Payment) < (DeliveryByCar | DeliveryByDrone)`

A typical training or validation file contains the system prompt corresponding to the full prompt given in Appendix A.1, a user prompt (i.e., an example of description), and an assistant prompt (i.e., the expressions corresponding to the description).

Appendix B

Fine-tuning Metrics

During its fine-tuning, GPT provides several metrics that one can analyse to understand how well the training is going. The two main metrics that we used to assess the quality of the resulting fine-tuned model were its accuracy and its loss. The accuracy, depicted in Figure B.1, shows how good the fine-tuned model is performing both on the training and the validation datasets. The closer its value is to 1, the better it is. On the other hand, the loss, depicted in Figure B.2, basically describes how different the output value of the model is from its expected output. Thus, the closer its value is to 0, the better it is.

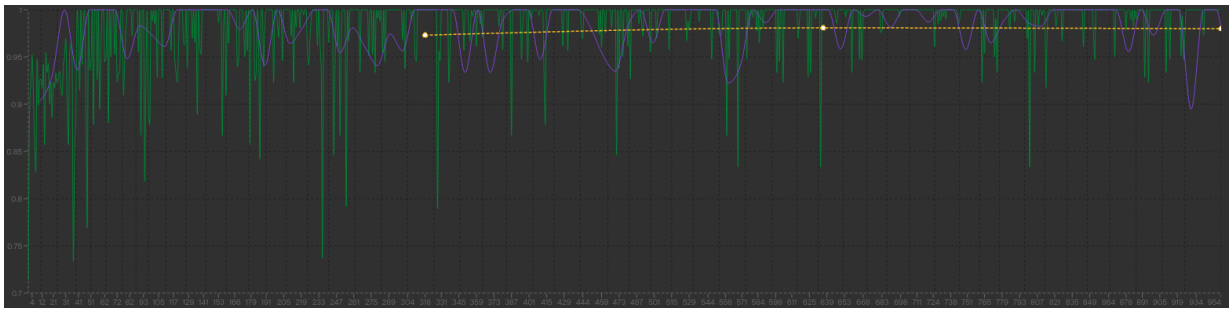


Figure B.1: GPT Fine-Tuning Accuracy over Training Iterations

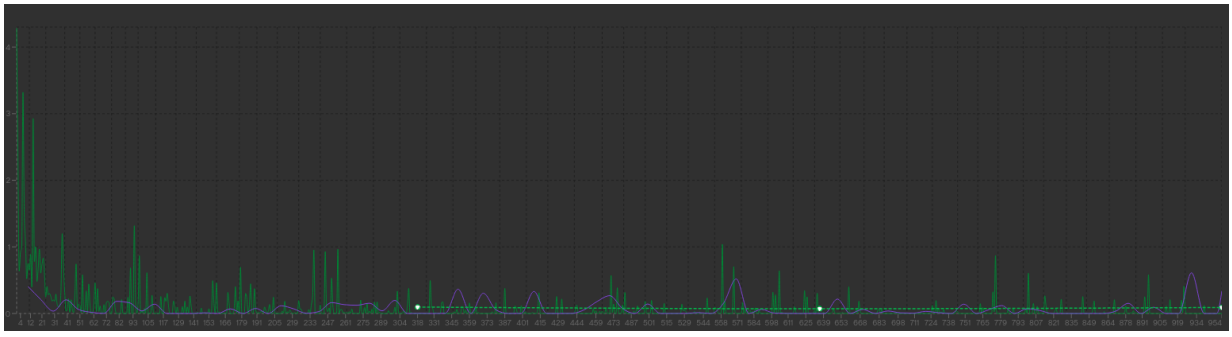


Figure B.2: GPT Fine-Tuning Loss over Training Iterations