


Master of Science in Informatics at Grenoble
Specialization Distributed Computing

Model checking and debugging of BPMN processes using coloration techniques

Quentin NIVON

June 27, 2022

Research project performed at INRIA - Team CONVECS

Under the supervision of:
Gwen SALAÜN

Defended before a jury composed of:
Yliès FALCONE
Martin HEUSSE
Thomas ROPARS

June 2022

Abstract

For decades, the business processes used by industries and companies have not stopped evolving and becoming more complicated. This had the consequence of enlarging the already existing gap between business analysts, developers and end users. Reducing this gap has been one of the main motivations for the BPMN notation to be invented. Business Process Model and Notation is a notation allowing to describe graphically a business process, with the goal of making it easily understandable for any type of users. In some companies, especially the ones dealing with safety-critical or safety-related systems, it is very important to perform verifications of the behaviour of the BPMN model(s) describing their process(es). Model checking is a well-known technique to perform such tasks, widely used since its beginnings in the 1980's. Indeed, if the property is verified in each point of the model, the model checker returns $True$. Otherwise it returns $False$ and gives a counterexample corresponding to a path in the specification that does not satisfy the property. Nonetheless, understanding this property can be complex, even for expert users. It is even more complex for beginner users, as the counterexample returned by the model checker is either textual (sequence of actions violating the property), or visual (graphical representation of the sequence of actions violating the property), but it is not expressed in the BPMN notation.

The challenges of this research project are **(i)** to give the most complete insights of the problem by taking into account all the counterexamples returned by the model checker, and **(ii)** to express them in the BPMN notation, ideally on the initial BPMN process, otherwise on a generated process, semantically equivalent to the initial one, and as syntactically close as possible to it. The approach presented in this work is a coloration approach, in which the nodes of the BPMN process are colored differently, according to their validity with regard to the property. Thus, each node is categorized as either *correct*, *incorrect*, or *neutral*. As one of our main goals is to perform the least changes on the initial BPMN process, we try first to color it directly, with the help of the counterexample(s) returned by the model checker. Nonetheless, for several reasons detailed in this report, coloring directly the initial BPMN process may not be possible. In such cases, we generate a new BPMN process semantically equivalent to the initial one. This new BPMN process has the advantage of being colorable regarding the counterexample(s) returned by the model checker. Once this new BPMN process is generated and colored, we try to reduce its number of nodes, while preserving its semantics, in order to make it more understandable for the user, and syntactically closer to the initial one.

In this report, we first present the background knowledge needed for its good understanding. Then, we describe precisely the followed approach. Third, we give an overview of the prototype built, along with some experiments. Finally, we present the related works and conclude with the future work.

Acknowledgement

I am very grateful to Professor Gwen SALAÜN for his advices during this internship, and for the time he spent answering my questions and rereading this

report.
I would also thank all the members of the CONVECS team of INRIA/LIG for making me feel comfortable during those months, and answering my questions.
Especially, I would like to thank Ajay KRISHNA MUROOR and Ahang ZUO for their help configuring, debugging and answering my questions regarding VBPMN, and Irman FAQRIZAL for his help understanding and using CLEAR, without whom this work would have been much more painful.
Last but not least, I would like to thank my family, especially my mother for her encouragement and her listening, and my father for always supporting me and giving me the taste of computer sciences.

Résumé

Depuis plusieurs dizaines d'années, les procédés métiers utilisés par les industries et les entreprises n'ont cessé d'évoluer et de se complexifier. Cela eut pour conséquence l'agrandissement du fossé déjà existant entre les analystes métier, les développeurs, et les utilisateurs finaux. C'est dans l'optique de réduire ce fossé qu'a été créée la notation BPMN. Le Business Process Model and Notation est une notation permettant de décrire graphiquement un procédé métier dans le but de le rendre compréhensible par tout type d'utilisateur. Pour certaines entreprises, notamment celles travaillant avec des systèmes critiques, il est très important d'effectuer des vérifications du comportement des modèles BPMN décrivant leurs procédés. La vérification de modèles est une technique connue, utilisée et fiable pour effectuer ce type de tâches, et ce depuis ses débuts dans les années 1980. En effet, si la propriété est vérifiée en tout point du modèle, le vérificateur de modèles retournera *True*. Si ce n'est pas le cas, celui-ci retournera *False* ainsi qu'un contre-exemple ne satisfaisant pas la propriété. Néanmoins, comprendre ce contre-exemple peut s'avérer complexe, même pour des utilisateurs avancés. Il l'est d'autant plus pour des utilisateurs néophytes, car le contre-exemple retourné par le vérificateur de modèles peut se trouver soit sous forme textuelle (séquence d'actions enfreignant la propriété), ou sous forme visuelle (représentation graphique de la séquence d'actions enfreignant la propriété), mais il n'est pas exprimé en notation BPMN.

Les enjeux de ce projet de recherche sont de **(i)** donner les informations les plus précises sur le problème en prenant en compte tous les contre-exemples retournés par le vérificateur de modèles, et de **(ii)** les représenter en notation BPMN, idéalement sur le procédé BPMN de départ, sinon sur un procédé BPMN généré, sémantiquement équivalent au procédé BPMN de départ, et aussi proche syntaxiquement de celui-ci que possible. L'approche proposée dans ces travaux est une approche par coloration, dans laquelle les noeuds du procédé BPMN sont colorés différemment, en fonction de leur validité au regard de la propriété. Ainsi, chaque noeud du procédé BPMN est catégorisé comme *correct*, *incorrect*, ou *neutre*. Sachant que l'un de nos objectifs principaux est d'effectuer le minimum de modifications syntaxiques du procédé BPMN de départ, nous essayons dans un premier temps de le colorer directement, avec l'aide des contre-exemples retournés par le vérificateur de modèles. Néanmoins, pour plusieurs raisons détaillées dans ce rapport, colorer directement le procédé BPMN de départ n'est pas toujours possible. Dans cette situation, nous générons un nouveau procédé BPMN sémantiquement équivalent au procédé BPMN de départ. Ce nouveau procédé à l'avantage d'être colorable vis-à-vis des contre-exemples retournés par le vérificateur de modèles. Une fois que ce nouveau procédé a été généré et coloré, nous essayons de réduire son nombre de noeuds, tout en préservant sa sémantique, dans le but de le rendre plus compréhensible pour l'utilisateur, et syntaxiquement plus proche du procédé BPMN de départ.

Dans ce rapport, nous présentons en premier lieu les connaissances nécessaires à sa compréhension. Ensuite, nous décrivons précisément l'approche proposée. Après cela, nous donnons une vue globale du prototype, accompagnée de plu-

sieurs expérimentations. Finalement, nous présentons les travaux connexes puis concluons avec le travail futur.

Contents

— 1 —
Introduction

For decades, the business processes used by industries and companies have not stopped evolving and becoming more complicated. This had the consequence of enlarging the already existing gap between business analysts, developers and end users. Reducing this gap has been one of the main motivations for the BPMN notation to be invented. Business Process Model and Notation is a notation allowing to describe graphically a business process, with the goal of making it easily understandable for any type of users. In some companies, especially the ones dealing with safety-critical or safety-related systems, it is very important to perform verifications of the behaviour of the BPMN model(s) describing their process(es). Model checking is a well-known technique to perform such tasks, widely used since its beginnings in the 1980's. Indeed, if the property is verified in each point of the model, the model checker returns $True$. Otherwise it returns $False$ and gives a counterexample corresponding to a path in the specification that does not satisfy the property. Nonetheless, understanding this property can be complex, even for expert users. It is even more complex for beginner users, as the counterexample returned by the model checker is either textual (sequence of actions violating the property), or visual (graphical representation of the sequence of actions violating the property), but it is not expressed in the BPMN notation.

This work strongly relies on some already existing techniques for converting BPMN into LNT [23], analyzing and debugging LNT specifications with respect to some temporal logic properties [29], and generating a colored LTS representing the full set of actions not satisfying the property, called a counterexample LTS (CLTS) [5]. But this CLTS can be hard to understand, for several reasons: **(i)** the CLTS notation strongly differs from the BPMN notation, **(ii)** the CLTS can be quite big if there are lots of bugs or if the BPMN contains certain types of constructs, such as parallel gateways, and **(iii)** the CLTS does not show exactly the source of the bugs.

In practice, the CLTS may give enough information regarding the bug(s) for expert users, but is not sufficient for beginner users, for whom the size of the CLTS and its syntactic differences with the BPMN notation may be a problem. To handle this problem, the main idea of the proposed approach is to perform an analysis of both the CLTS and the BPMN files. This analysis is separated in 2 phases. The first one, called *matching analysis*, aims at verifying whether the satisfaction/violation of the property can be directly represented on the initial BPMN process. If this is the case, we color this BPMN process and give it back to the user. This is somehow the best case, because no modification of the initial BPMN process is required. If this can not be done, we generate a new BPMN process. This generated process is semantically

equivalent to the initial one. Moreover, it has, by construction, the advantage of allowing us to color it to represent the satisfaction and the violation of the property. However, this generated BPMN process can possibly be quite large, in terms of number of nodes. In this case, understanding this process can be difficult for the user. To limit this, we propose a solution in which we try to reduce the size of this BPMN process, while preserving its semantics. We called this phase *folding*.

It is important to notice that the class of temporal properties handled in this work is the class of safety properties, which is widely used in the verification of real-time and critical systems. This class of properties covers behaviours of type *Something bad must not happen*.

Figure 1.1 gives an overview of the contributions presented in this report. The approach takes as input the initial BPMN process and the temporal logic property to verify. The very first step consists in giving the BPMN process to the VBPMN tool, which translates it into LNT. Then, the LNT specification and the temporal logic property are given to CLEAR [6], which generates a CLTS. Starting from this CLTS and the BPMN process, we perform the matching analysis. If a matching is found between the CLTS and the BPMN process, we perform coloration. Otherwise, we generate a new BPMN process with the help of the CLTS (*Unfolding*), we try to fold it, and we color it. In addition to the algorithmic approach proposed, we also present the prototype implementing the different presented solutions, and 2 experimental studies. In the first one, we confront our approach to various real-world examples to get insights of its behaviour and of the quality of its results. The second one focuses on the time taken by the approach to perform on various examples. In this second study, the goal is to assess the scalability of the approach on real-world examples, and its limits in terms of number of nodes of a BPMN process.

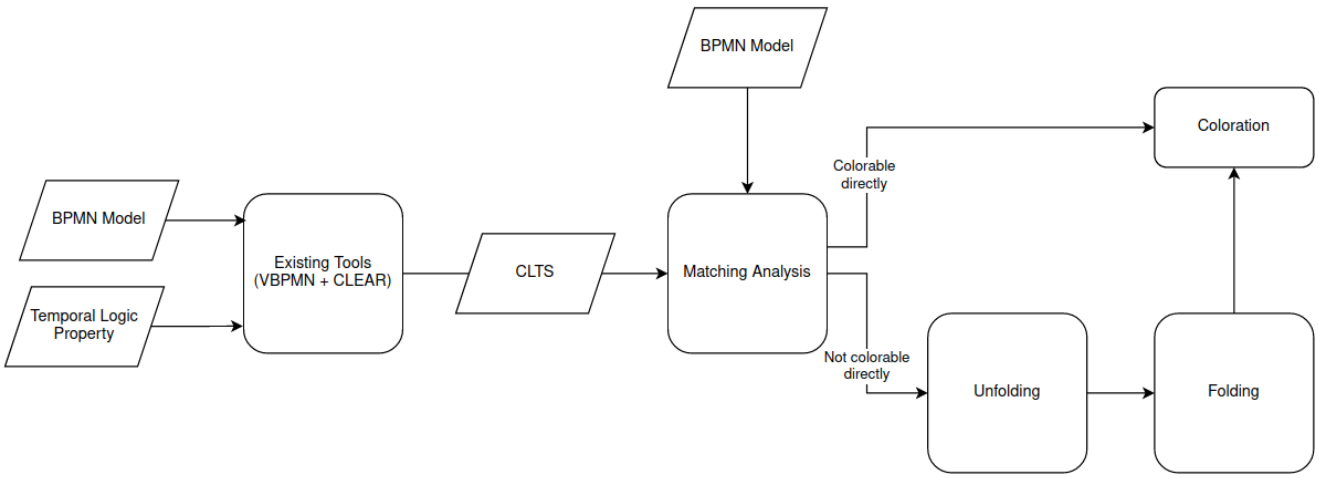

Figure 1.1: Overview of the approach

This research project's report contains the following chapters:

- **Chapter 2** describes the background knowledge needed to understand and accomplish this work.

- **Chapter 3** explains the contributions, by detailing the approach and the algorithms.
- **Chapter 4** contains implementation details and experimentation made to evaluate the work.
- **Chapter 5** surveys related work.
- **Chapter 6** concludes this report and proposes some perspectives to the approach.

— 2 —
Background

In this chapter, we focus on some notations, needed to achieve this work and for its good understanding. First we describe our input model, BPMN, by detailing its purpose and its syntax, and presenting briefly the VBPMN tool [23]. Then, we give an overview of Labelled Transition Systems and temporal properties, especially safety properties. Finally, we explain what is a counterexample LTS (CLTS), and describe how the CLEAR tool produces it.

2.1 BPMN

2.1.1 Overview

Business Process Model and Notation (BPMN) is a business process modeling method aiming at describing business processes used by industries and companies. The main goal of this notation is to provide a way of representing precisely the process used by the company, while in the mean time being easy to understand by any user of the company, from the business analysts, to the developers and finally the end users. This notation has been first introduced in 2004 by the Business Process Management Initiative (BPMI) and has become an ISO/IEC standard in 2013 [3]. It is now widely used across the world. The current version of BPMN is BPMN 2.0. Its exact specifications can be found in [3].

2.1.2 Syntax

The BPMN notation follows a precise syntax, that the reader can find detailed in [3]. It is mainly composed of events, activities, gateways and flows. In this work, we focus on a subset of this full syntax described in Figure 2.1. This subset takes into account the initial and end events that respectively start and end a BPMN process, the tasks, that represents steps of the process, and the gateways, either exclusive, parallel or inclusive, that represent choices. Considering the whole syntax would be interesting, but it highly increases the complexity of the work. Moreover, our subset suffice to cover more than 90% of BPMN processes, according to a study made in [22] on more than 800 BPMN processes.

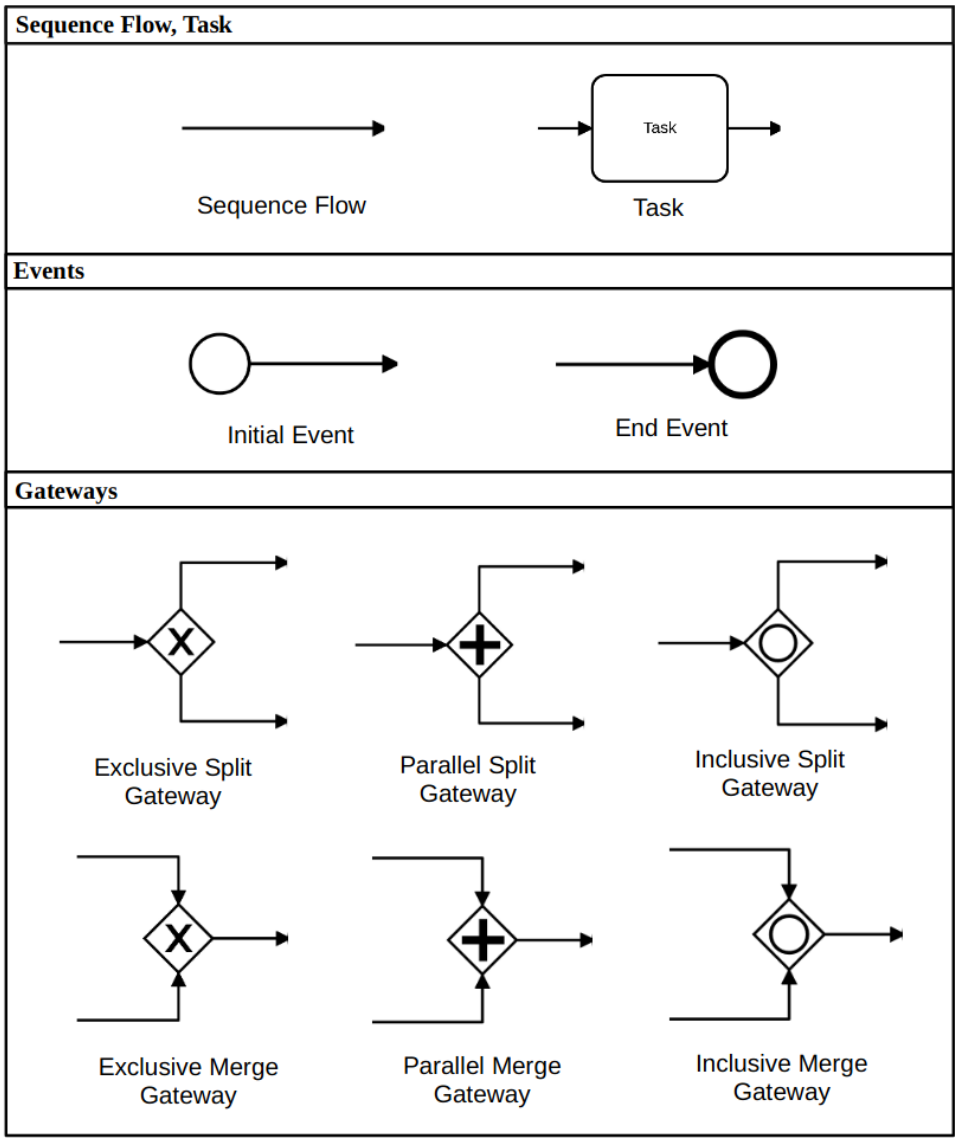

Figure 2.1: Supported BPMN Syntax

It is interesting to note that the gateways are presented in pairs (split gateway, merge gateway) in this figure. The reader can see the split gateway as the beginning of a choice, and the merge gateway as the end of this choice. Such gateways are called *balanced gateways*. Nonethelees, the BPMN notation allows one not to put the merge gateway, while preserving the BPMN syntax. Split gateways without their corresponding merge gateways are called *non-balanced gateways*.

From this syntax, one is able to generate a BPMN process representing almost any kind of business process. With only the subset of the BPMN syntax presented in Figure 2.1, one can model complex realistic examples, such as the account opening process depicted in Figure 2.2, which will be our running example.

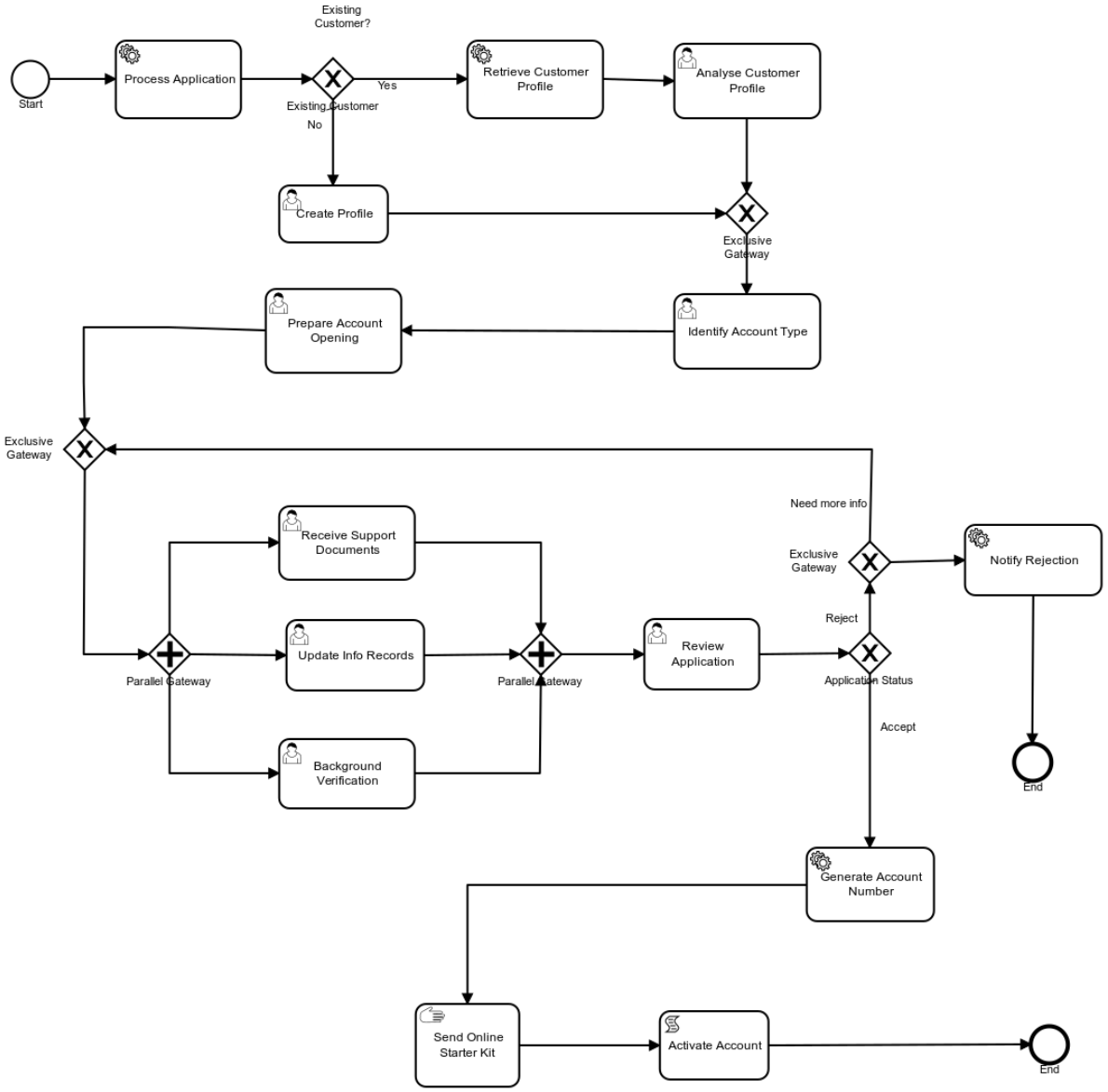

Figure 2.2: Account Opening Process in BPMN

For the rest of this work, a BPMN process B will be considered as a graph, composed of nodes (tasks, gateways, events) and edges (sequence flows) connecting these nodes.

Definition 1. *(BPMN Process) Let S_N be a set of nodes, being either BPMN tasks, BPMN gateways or BPMN events, and S_E a set of edges, being BPMN sequence flows, connecting these nodes. Any BPMN process B compliant to the syntax presented in Figure 2.1 can be written as $B = (S_N, S_E)$.*

2.1.3 VBPMN

Now that we are able to represent business processes in BPMN, we would like to be able to verify temporal properties over it. For this purpose, we use model checking techniques. The problem is that current model checkers do not accept BPMN processes as input specifications.

To overcome that, we make use of the VBPMN tool [23] to translate BPMN into LNT. Then, the LNT specification and the temporal logic property can be given to a model checker for verification.
VBPMN is a tool created in 2017, aiming at verifying properties over BPMN processes and comparing BPMN processes between them [23]. It is freely accessible through any navigator with the use of a Tomcat server, by following the instructions available in [2]. Figure 2.3 shows the web interface offered by VBPMN to verify properties.

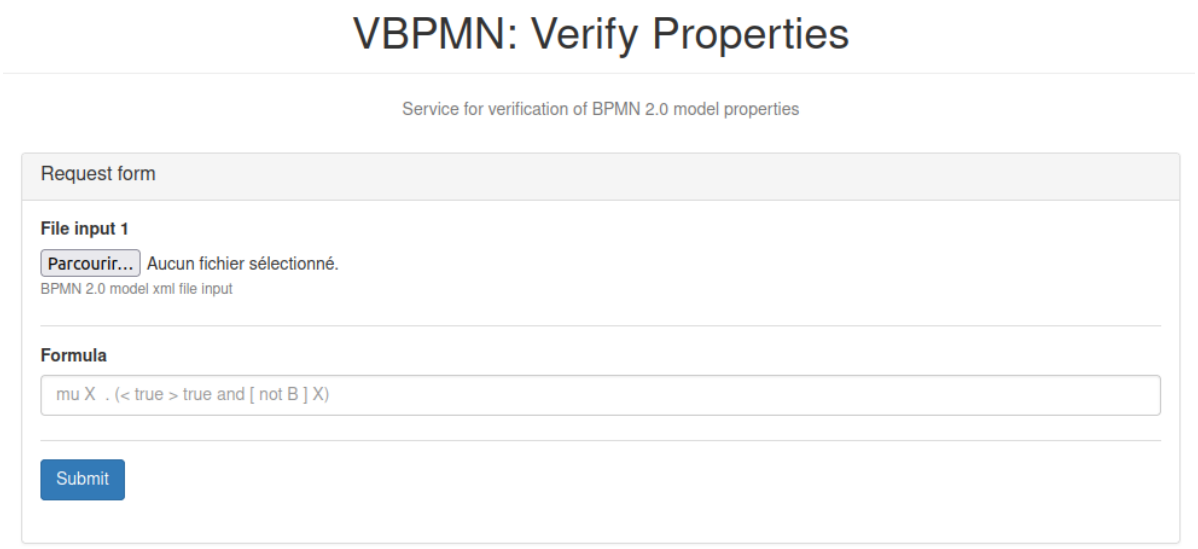

Figure 2.3: Web interface of the VBPMN verification tool

In this work, we make use of VBPMN to translate BPMN into LNT. According to [23], the BPMN process is first translated into an intermediate format called PIF. This PIF file is then verified to make sure that its semantics conforms to the semantics of the BPMN process. Then, in a second time, the PIF file is translated into LNT, and the same kind of verification is performed to make sure that the semantics of the generated LNT file conforms to the semantics of the initial BPMN process. By doing this, we ensure a semantical equivalence between the initial BPMN process and the generated LNT specification.
The LNT specification generated by VBPMN is then compiled into LTS using the CADP toolbox [18].

2.2 Labelled Transition Systems (LTS)

In this work, LTS is used as input for the model checker, and, in a slightly different version called CLTS, used as one of the inputs of the proposed approach. LTSs are used by a large set of verification tools, such as CADP [18], mCRL2 [13], LTSmin [20] or TAPAs [9]. An LTS is represented by a set of states, and a set of transitions connecting these states. In CADP [18], which we use for this work, it is encoded in the AUT format. In this format, the first line gives information about the initial state, the total number of states, and the total number of transitions, while all the other lines describe the transitions of the system.

Definition 2. *(LTS) An LTS is a tuple $M = (S, s^0, \Sigma, T)$ where S is a finite set of states, $s^0 \in S$ is the initial state, Σ is a finite set of labels, $T \subseteq S \times \Sigma \times S$ is a finite set of transitions.*

A transition is represented as $s \xrightarrow{l} s' \in T$, where $l \in \Sigma$. An LTS is usually produced from a higher-level specification of the system described with a process algebra (here, LNT). Such specification can be compiled into an LTS using specific compilers (CADP [18] for instance). An LTS exhibits all possible executions of a system. One specific execution is called a *trace*.

Definition 3. *(Trace) Given an LTS $M = (S, s^0, \Sigma, T)$, a trace of size $n \in \mathbb{N}$ is a sequence of labels $l_1, l_2, \ldots, l_n \in \Sigma$ such that $s^0 \xrightarrow{l_1} s_1 \in T, s_1 \xrightarrow{l_2} s_2 \in T, \ldots, s_{n-1} \xrightarrow{l_n} s_n \in T$. A trace is either infinite because of loops or the last state s_n has no outgoing transitions. The set of all traces of M is written as $t(M)$.*

Note that $t(M)$ is prefix closed. One may not be interested in all traces of an LTS, but only in a subset of them. To this aim, we introduce a particular label δ, called *final label*, which marks the end of a trace, similarly to the notion of accepting state in language automata. This leads to the concept of *final trace*.

Definition 4. *(Final Trace) Given an LTS $M = (S, s^0, \Sigma, T)$, and a label δ, called final label, a final trace is a trace $l_1, l_2, \ldots, l_n \in \Sigma$ such that $s^0 \xrightarrow{l_1} s_1 \in T, s_1 \xrightarrow{l_2} s_2 \in T, \ldots, s_{n-1} \xrightarrow{l_n} s_n \in T$, $l_1, l_2, \ldots, l_n \neq \delta$ and there exists a final transition $s_n \xrightarrow{\delta} s_{n+1}$. The set of final traces of M is written as $t_\delta(M)$.*

Note that the final transition characterized by δ does not occur in the final traces and that $t_\delta(M) \subseteq t(M)$. Moreover, if M has no final label then $t_\delta(M) = \emptyset$.

2.3 Temporal Properties

Model checking consists in verifying that an LTS model satisfies a given temporal property φ, which specifies some expected requirement of the system. Temporal properties are usually divided into two main families: *safety* and *liveness* properties [29]. However, in this work, we focus on safety properties. Managing liveness properties is part of future work.

Safety Properties

Safety properties are widely used in the verification of real-world systems, and especially in safety-critical and safety-related systems. These properties state that "*something bad never happens*". A safety property is usually formalised using a temporal logic. In this work, the temporal logic properties are written in Model Checking Language (MCL) [26]. MCL is an action-based, branching-time temporal logic used in order to express concurrent systems temporal properties. A safety property can be semantically characterised by an infinite set of traces t_φ, corresponding to the traces that violate the property φ in an LTS. If the LTS model does not satisfy the property, the model checker returns a *counterexample*, which is one of the traces characterised by t_φ.

Definition 5. *(Counterexample for Safety Properties) Given an LTS $M = (S, s^0, \Sigma, T)$ and a safety property φ, a counterexample is any trace which belongs to $t(M) \cap t_\varphi$.*

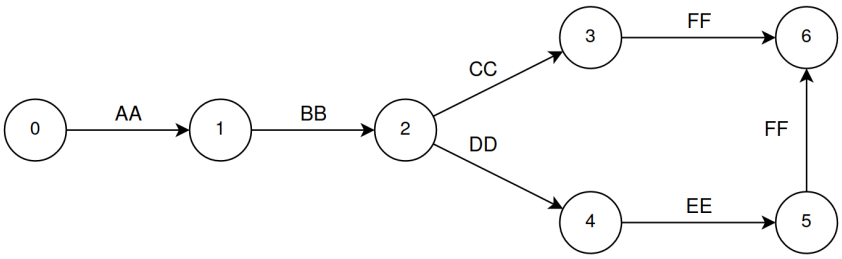

Figure 2.4: An LTS example

Figure 2.4 presents a simple example of LTS consisting of several actions in sequence and of a choice in State 2. We give in Table 2.1 some examples of safety properties, accompanied with their counterexamples. For instance, we can look at the property [*true** . *"AA"* . *true** . *"DD"* . *true**] *false*. By studying the LTS in Figure 2.4, we can see that the property is violated if we chose to go from state 2 to state 4. Starting from the initial state, one can find the following path violating the property: *AA, BB, DD*, which is the the counterexample proposed in the table.

Safety property	Counterexample
[true* . "AA" . true* . "DD" . true*] false	AA, BB, DD
[true* . "AA" . true* . "DD" . true* . "FF" . true*] false	AA, BB, DD, EE, FF
[(not "BB")* . "AA" . true*] false	AA
[true* . "AA" . (not "DD")* . "CC" . true*] false	AA, BB, CC

Table 2.1: Example of safety properties with their corresponding counterexamples

In this work, we will focus on properties of form [*true** . *"T_1"* . *true** . *"..."* . *true** . *"T_n"* . *true**] *false* and refer to the tasks $(T_1, ..., T_N)$ belonging to the property as *property tasks*.

2.4 Counterexample LTS (CLTS)

2.4.1 Overview

The approach presented in [5] takes as input an LNT specification, which compiles into an LTS model, and a temporal property. The original idea of this work is to identify decision points where the specification (and the corresponding LTS model) goes from a (potentially) correct behaviour to an incorrect one. These choices turn out to be very useful to understand the source of the bug. These decision points are called *faulty states* in the LTS model.
In order to detect these faulty states, we first need to categorize the transitions in the model into different types. The transition type allows to highlight the compliance with the property of the paths in the model that traverse that given transition. Transitions in the counterexample LTS can be categorized into three types:

- *correct transitions*, which belong to paths in the model that represent behaviours which always satisfy the property.
- *incorrect transitions*, which belong to paths in the model that represent behaviours which always violate the property.

- *neutral transitions*, which belong to portions of paths in the model which are common to correct and incorrect behaviours.

The information concerning the detected transitions type (correct, incorrect and neutral transitions) is added to the initial LTS in the form of tags. The set of transition tags is defined as $\Gamma = \{correct, incorrect, neutral\}$. Given an LTS $M = (S, s^0, \Sigma, T)$, a tagged transition is represented as $s \xrightarrow{(l,\gamma)} s'$, where s, $s' \in S$, $l \in \Sigma$ and $\gamma \in \Gamma$. Thus, an LTS in which each transition has been tagged with a type is called *tagged LTS*.

Definition 6. *(Tagged LTS) Given an LTS $M = (S, s^0, \Sigma, T)$, and the set of transition tags Γ, the tagged LTS is a tuple $M_T = (S_T, s^0_T, \Sigma_T, T_T)$ where $S_T = S$, $s^0_T = s^0$, $\Sigma_T = \Sigma$, and $T_T \subseteq S_T \times \Sigma_T \times \Gamma \times S_T$.*

The tagged LTS where transitions have been typed allows us to identify faulty states in which an incoming neutral transition is followed by a choice between at least two transitions with different types (correct, incorrect, neutral). Such a faulty state consists of all the neutral incoming transitions and all the outgoing transitions.

Definition 7. *(Faulty State) Given the tagged LTS $M_T = (S_T, s^0_T, \Sigma_T, T_T)$, a state $s \in S_T$, such that $\exists t = s' \xrightarrow{(l,\gamma)} s \in T_T$, t is a neutral transition, and $\exists t' = s \xrightarrow{(l,\gamma)} s'' \in T_T$, t' is a correct or an incorrect transition, the faulty state s consists of the set of transitions $T_{nb} \subseteq T_T$ such that for each $t'' \in T_{nb}$, either $t'' = s' \xrightarrow{(l,\gamma)} s \in T_T$ or $t'' = s \xrightarrow{(l,\gamma)} s''' \in T_T$.*

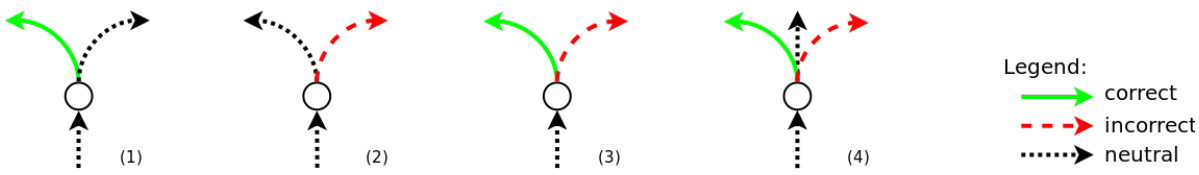

Figure 2.5: The Four Types of Faulty States

By looking at outgoing transitions of a faulty state, we can identify four categories of faulty states (Figure 2.5):

1. with at least one correct transition and one neutral transition (no incorrect transition),
2. with at least one incorrect transition and one neutral transition (no correct transition),
3. with at least one correct and one incorrect transition (no neutral transition), and
4. with at least one correct, one incorrect, and one neutral transition.

2.4.2 CLEAR

CLEAR is a tool presented in [6], which is available online [1]. The tool architecture is depicted in Figure 2.6. It consist of three main modules: **(i)** faulty state calculation module, **(ii)** 3D visualization module and **(iii)** analysis module.

The faulty state calculation module is responsible for the transition types recognition and for the faulty states computation. The input of this module is the LNT specification and a temporal property. It partially relies on the CADP toolbox [18]. The calculation module is divided into three components. The first one implements the Counterexample LTS approach

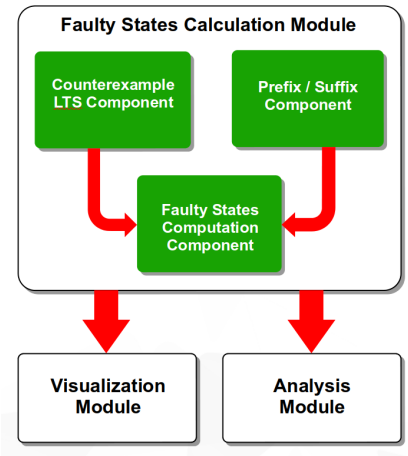

Figure 2.6: Overview of CLEAR Tool

for computing the tagged LTS for safety properties. The second one implements the Prefix / Suffix approach for computing the tagged LTS for liveness properties. The third component allows computing the faulty states in the tagged LTS, and is common to both approaches. The result of this model is the counterexample LTS.

The CLEAR visualization techniques [7] give the developer a graphical representation of the counterexample LTS extended with faulty states, where correct/incorrect/neutral transitions and faulty states are highlighted. It is a web based application which visualizes the counterexample LTS in 3D. The techniques make use of different colours to distinguish correct (green), incorrect (red) and neutral (black) transitions on the one hand, and all kinds of faulty states (represented with 4 different shades of yellow) on the other hand. Moreover, the user can exploit various functionalities to better inspect the faulty model: forward/backward step-by-step animation, counterexample visualization, zoom in/out on the faulty states, etc. The goal of this visual rendering is to provide a support for visualizing the erroneous part of the tagged LTS. This visualization emphasizes all the faulty states where a choice is taken and makes the specification either head to a correct or an incorrect behaviour. Figure 2.7 presents the interface of the visualization techniques.

The CLEAR analysis module provides the implementation of abstraction techniques. It first produces the counterexample from an LTS and a property using the CADP model checker. Second, it performs the counterexample reduction by locating and keeping actions that correspond to faulty states, by comparing states of the counterexample to the ones belonging to the counterexample LTS. The result retrieved by this technique consists of a counterexample abstracted in the form of a list of sub-sequences of actions, accompanied by the list of all faulty states.

For a better understanding of this report, screenshots of CLEAR (such as Figure 2.7) will be replaced by an equivalent version that is more understandable and clearer than simple screenshots. Still, we will refer to it as CLEAR outputs. For example, the screenshot of CLEAR in Figure 2.8 will be replaced by its equivalent representation in Figure 2.9, where correct, incorrect and neutral states/transitions follow the same representation rules as the ones proposed in Figure 2.5.

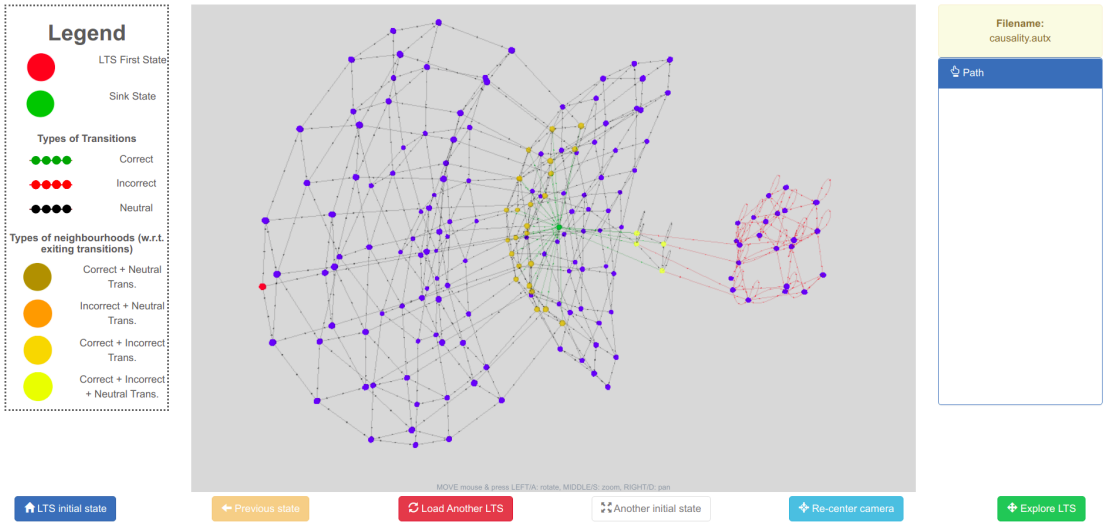

Figure 2.7: The interface of the CLEAR visualization module

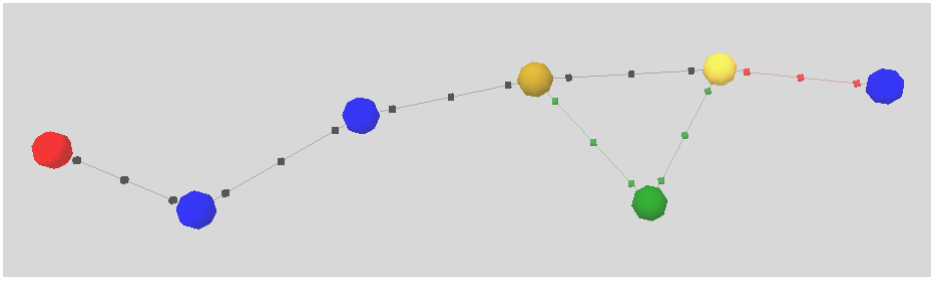

Figure 2.8: Screenshot of the CLEAR visualization tool

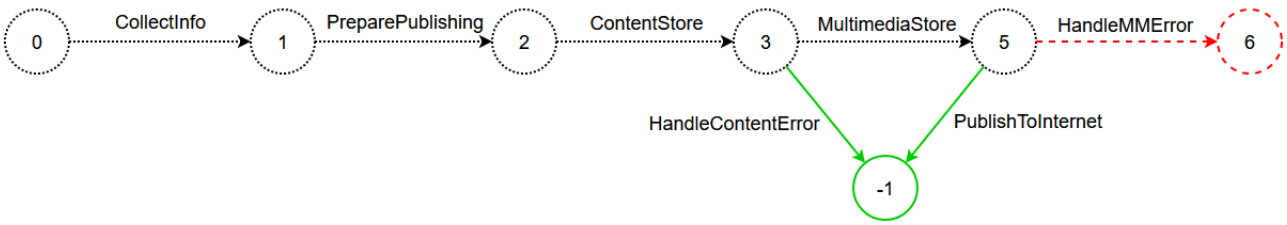

Figure 2.9: Clearer version of the screenshot in Figure 2.8

— 3 —
Contributions

For now, existing techniques used to debug BPMN processes give either textual explanations (sequence(s) of actions violating the property, written in plain text) or visual explanations (sequence(s) of actions violating the property, presented visually (LTS, CLTS)). The main problems of these techniques remain **(i)** the syntactic differences between the counterexample and the BPMN notation, **(ii)** the possibly huge size of the counterexample returned, and **(iii)** the lack of understanding of the problem's source.
The goal of this approach is to go one step further than the existing solutions by expressing the counterexamples of the property directly in BPMN notation. As we also aim at reducing the gap between the counterexample(s) returned and the initial BPMN process, we first try to represent the counterexample(s) on the initial BPMN. This is possible if and only if a mapping between all the faulty states of the CLTS and some BPMN nodes can be found. We called the detection of this mapping the *matching analysis*. If such a mapping does not exist, we generate a new BPMN process, semantically equivalent to the initial, that allows us to represent the counterexample(s) of the property. Then, we try to fold this generated process to make it more readable and more understandable for the user, and syntactically closer to the initial one.
The matching analysis is detailed in Section 3.1 while Section 3.2 describes the generation of the new BPMN process and the folding phase. To conclude this chapter, Section 3.3 presents additional ideas that have been discussed and are still under development.

3.1 Matching Analysis & Direct Coloration

This section consists of two main parts, the first one concerning the matching analysis – which verifies if the initial BPMN diagram is directly colorable with respect to the given property –, and the second one concerning the direct coloration of the initial BPMN process, when a matching is found. Before describing the matching analysis, let us define some notions that are used all along this report.

3.1.1 Definitions

In BPMN processes, nodes are connected between them by sequence flows. One may be interested in knowing if, starting from a given node, a sequence of BPMN nodes can lead to another given node. This is the notion of *path* between two nodes.

Definition 8. *(Path) Let $B = (S_N, S_E)$ be a BPMN process. Let E_1, $E_2 \in S_N$. A path p between E_1 and E_2 is a sequence of BPMN nodes $(e_1, ..., e_n)$ s.t. $E_1 = e_1$, $E_2 = e_n$ and $\forall i \in [1...(n-1)], \exists s_i \in S_E$ s.t. e_i is connected to e_{i+1} with s_i.*

From this definition, we define the *size* of a path.

Definition 9. *(Path Size) Let $p = (e_1, ..., e_n)$ be a path between two BPMN nodes E_1 and E_2. The size of p is defined as $size(p) = |(e_1, ..., e_n)|$.*

At some point of this work, we will need to compute the set of *outgoing paths* of a node. This set represents all the possible paths that can be taken from a node, and which finish either by an element that is already in the path (loop), or by an end event. To do so, we perform a depth-first graph traversal, in which we pay a particular attention to cycles by keeping in a list all the nodes that have already been visited. In the next sections, we will consider the $Out(E)$ operator, which computes the set of outgoing paths of any BPMN node E.
From this definition, we derived the set of *inner paths* of a split gateway. In this set, if the split gateway considered is balanced, each path ends with its corresponding merge gateway, or by an element that is already in the path (loop). If the split gateway is non-balanced, then each path ends with an end event, or by an element that is already in the path (loop). In this second case, the set of outgoing paths and the set of inner paths are equivalent. In the next sections, we will consider the $Inner(G_S)$ operator, which computes the set of inner paths of any split gateway G_S.

Starting from the set of outgoing paths of a node, we can consider a set containing only the first node of each outgoing paths. This set contains the *children* of a node.

Definition 10. *(Children of a Node) Let $B = (S_N, S_E)$ be a BPMN process and $E \in S_N$ a node. We call $Child(E) \subset S_N$ the set of children of E, s.t. $\forall C_E \in Child(E)$, $\exists s \in S_E$ s.t. E is connected to C_E with s.*

Finally, we define the size of a split gateway as its number of children.

Definition 11. *(Gateway Size) Let $B = (S_N, S_E)$ be a BPMN process, $G \in S_N$ be a split gateway. The size of G is defined as its number of children, e.g. $size(G) = |Child(G)|$.*

3.1.2 Matching Analysis

One of the main concerns of this work is to return to the user a BPMN process that is as syntactically close as possible to the one taken as input, while remaining semantically equivalent. The best way to stick to this constraint is to return to the user the initial BPMN process, where the only difference is its coloration. To be able to perform this direct coloration, 2 conditions need to be satisfied. First, the transitions of the CLTS having the same label need to be colored identically. More formally, *if $M_T = (S_T, s_T^0, \Sigma_T, T_T)$ is a CLTS, M_T has valid transitions color if and only if $\nexists\, t_1, t_2 \in T_T$ s.t. $label(t_1) = label(t_2)$ and $color(t_1) \neq color(t_2)$.*
Such a case can happen for various reasons. One of them comes from the syntactic differences between the BPMN notation and the LTS notation. Indeed, one can represent parallelism in BPMN notation (parallel gateway) but can not in LTS notation. In fact, the LTS notation only supports sequential execution. According to Milner's expansion theorem [4], parallelism can be rewritten using choice and sequentiality. This means that the LTS representation of a BPMN process containing parallel gateways will contain choice and sequentiality to represent

this parallelism. In this case, it may happen that the first outgoing path of the choice satisfies the property, while the second one violates it. But in such case, the initial BPMN process can not be colored directly, as these correct and incorrect paths do not appear in it.
Second, we need to have a *matching* between the initial BPMN process and the CLTS. This notion of matching relies on the notion of *directly reachable task*. A task is said to be directly reachable from a node if it exists a path between this node and the task which does not contain any other task.

Definition 12. *(Directly Reachable Task) Let $B = (S_N, S_E)$ be a BPMN process, and $S_{T_N} \subset S_N$ the set of BPMN tasks of B. $\forall t_N \in S_{T_N}, t_N$ is directly reachable from $n \in S_N$ if and only if there exists a path $p = (e_1, ..., e_n)$ where $n = e_1$, $t_N = e_n$, and $\forall i \in [2...(n-1)]$, $e_i \notin S_{T_N}$.*

If the first condition is satisfied, the matching analysis is performed using Algorithm 1. In this algorithm, we verify whether there exists an injection between the set of faulty states of the CLTS, and the set of BPMN nodes of the initial BPMN process. The core of the algorithm resides in the function *FindCorrespondingNode*. This function is a recursive function, which is called on each faulty state. Its role is to verify whether the current faulty state can be associated to one of the BPMN nodes composing the BPMN process, starting from the initial node. To do so, we check whether the outgoing (resp. incoming) transitions of the current faulty state have the same label than the outgoing (resp. incoming) directly reachable tasks of the current BPMN node. If this is the case, and if the current BPMN node has not already been associated to a faulty state, we store this association, and mark the current BPMN node as used, so that it can not be associated to any other faulty state. Then, we restart the process on the next faulty state, if it exists. Otherwise, if no association was found between the current faulty state and the current BPMN node, we call recursively this function on the same faulty state, and all the children of the current BPMN node. If after traversing the whole BPMN process, no association could be found with the current faulty state, the current faulty state is marked as *non matchable*, and we know that the BPMN process can not be colored directly. Otherwise, if all the faulty states have been linked to unique BPMN nodes, we say that we have a matching between the CLTS and the BPMN process, and direct coloration is performed. We represent a matching between a BPMN process B and a CLTS M with the operator *Match(B,M)*.

Algorithm 1 Matching Analysis Algorithm

```
1:  procedure MATCHES
2:      S_FS ← Set of CLTS faulty states
3:      NonMatchableNodes, MatchableNodes ← ∅
4:      InitialNode ← Graph initial node
5:      for all fs ∈ S_FS do
6:          CorrespondingNode ←FINDCORRESPONDINGNODE(fs, InitialNode)
7:          if CorrespondingNode ≠ null then
8:              MatchableNodes ← MatchableNodes ∪ {CorrespondingNode}
9:          else
10:             NonMatchableNodes ← NonMatchableNodes ∪ {CorrespondingNode}
11:         end if
12:     end for
13:     return NonMatchableNodes == ∅
14:
15:     function FINDCORRESPONDINGNODE(FaultyState, CurrentNode)
16:         T_Out ← Set of faulty state's outgoing transitions
17:         T_Inc ← Set of faulty state's incoming transitions
18:         Match ← T_Out is empty
19:         for all transition ∈ T_Out do
20:             Match ← False
21:             for all child of CurrentNode do
22:                 if child.label = transition.label then Match ← True
23:             end for
24:             if Match = False then break
25:         end for
26:
27:         if Match = True then
28:             for all transition ∈ T_Inc do
29:                 Match ← False
30:                 for all parent of CurrentNode do
31:                     if parent.label = transition.label then Match ← True
32:                 end for
33:                 if Match = False then break
34:             end for
35:         end if
36:
37:         if Match = True and CurrentNode has not already been linked then
38:             return CurrentNode
39:         else
40:             for all child of CurrentNode do
41:                 CorrespondingNode ← FINDNODE(fs, child)
42:                 if CorrespondingNode ≠ null then return CorrespondingNode
43:             end for
44:         end if
45:
46:         return null
47:     end function
48: end procedure
```

In the worst case, this algorithm traverses the whole BPMN graph once for each faulty state of the CLTS. As the complexity of a full graph traversal is linear, the whole complexity of this algorithm remains linear after multiplying it by the number of faulty states of the full CLTS. More precisely, if S_N is the set of nodes of the initial BPMN graph, and S_{FS} the set of faulty states of the CLTS, then Algorithm 1 has a worst-case complexity of $O(|S_N||S_{FS}|)$. $|S_{FS}|$ being constant, the worst-case complexity can be simplified to $O(|S_N|)$, which is linear.

Now, let us illustrate this algorithm with the BPMN process given in Figure 2.2 and the property *I do not want to notify rejection* translated into MCL as: [*true** . *Notify Rejection* . *true**] *false*.

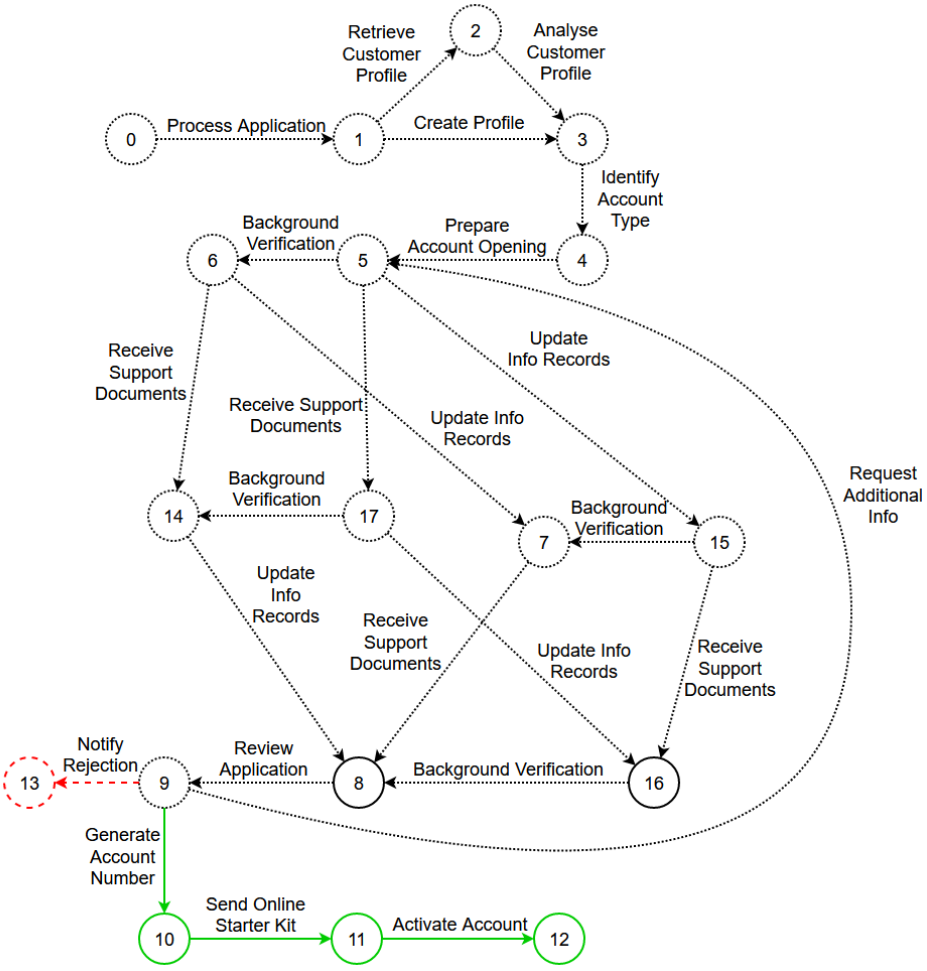

Figure 3.1: Full CLTS for the Account Opening example

Starting from the CLTS returned by CLEAR shown in Figure 3.1 and the BPMN process presented in Figure 2.2, we verify if all the transitions having the same label have the same color. If it is the case, we perform the matching analysis. The result of this verification is given in Table 3.1. Here, the condition is satisfied because transitions labels appearing in each column are

distinct, meaning that there is no transitions with the same label having two (or more) different colors. In such case, we can go to the next step: the matching analysis.

Red transitions	**Green transitions**	**Black transitions**
Notify Rejection	Generate Account Number	Process Application
	Send Online Starter Kit	Retrieve Customer Application
	Activate Account	Analyse Customer Profile
		Create Profile
		Identify Account Type
		Prepare Account Opening
		Receive Support Documents
		Update Info Records
		Background Verification
		Review Application
		Request Additional Info

Table 3.1: Results of transitions color categorization

The next step of the procedure consists in associating to each faulty state of the CLTS (here 1: state 9) a unique node of the BPMN process. This is the *matching* phase. In this case, we see in Figure 3.2 that the unique faulty state (isolated on the Figure) has been successfully matched to a node of the BPMN process (isolated on the Figure). As each faulty state has been successfully linked to a unique element of the BPMN diagram, direct coloration can be performed.

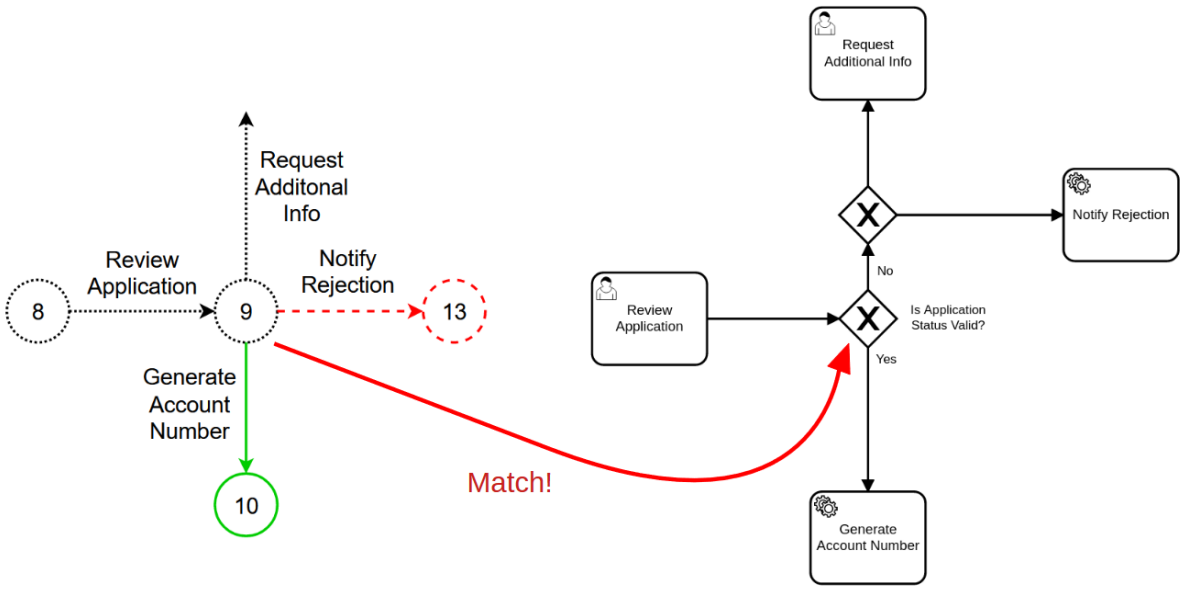

Figure 3.2: Correspondence between faulty state and BPMN element found

3.1.3 Direct Coloration

Once the matching analysis has returned *True*, meaning that we can color directly the initial BPMN process without modifying it, the procedure is the following: for each faulty state of the CLTS, we retrieve its matching node in the BPMN process. Then, for each red outgoing transition of the faulty state, we color in red the directly reachable task of the matching node that has the corresponding label. We repeat the same procedure for each green transition of the faulty state.
Now that we have colored all the directly reachable tasks of the matching node in red (resp. green), we color in red (resp. green) all the descendant tasks of these tasks. To do so, we perform a reachability analysis, starting from each directly reachable task of the matching node. This analysis consists in a depth-first graph traversal in which all the nodes of the BPMN process that are tasks are retrieved to be colored afterwards. During this traversal, specific attention is paid to cycles, in order to avoid unbounded recursion. To do so, each visited node is stored in a list of already visited nodes, and if the function performing the reachability analysis is called on a node already visited, it stops immediatly. Once we have retrieved all the reachable tasks for each directly reachable task of the matching node, we color them in the same color than the current directly reachable task.
Figure 3.3 shows the result of this direct coloration on the previous example.

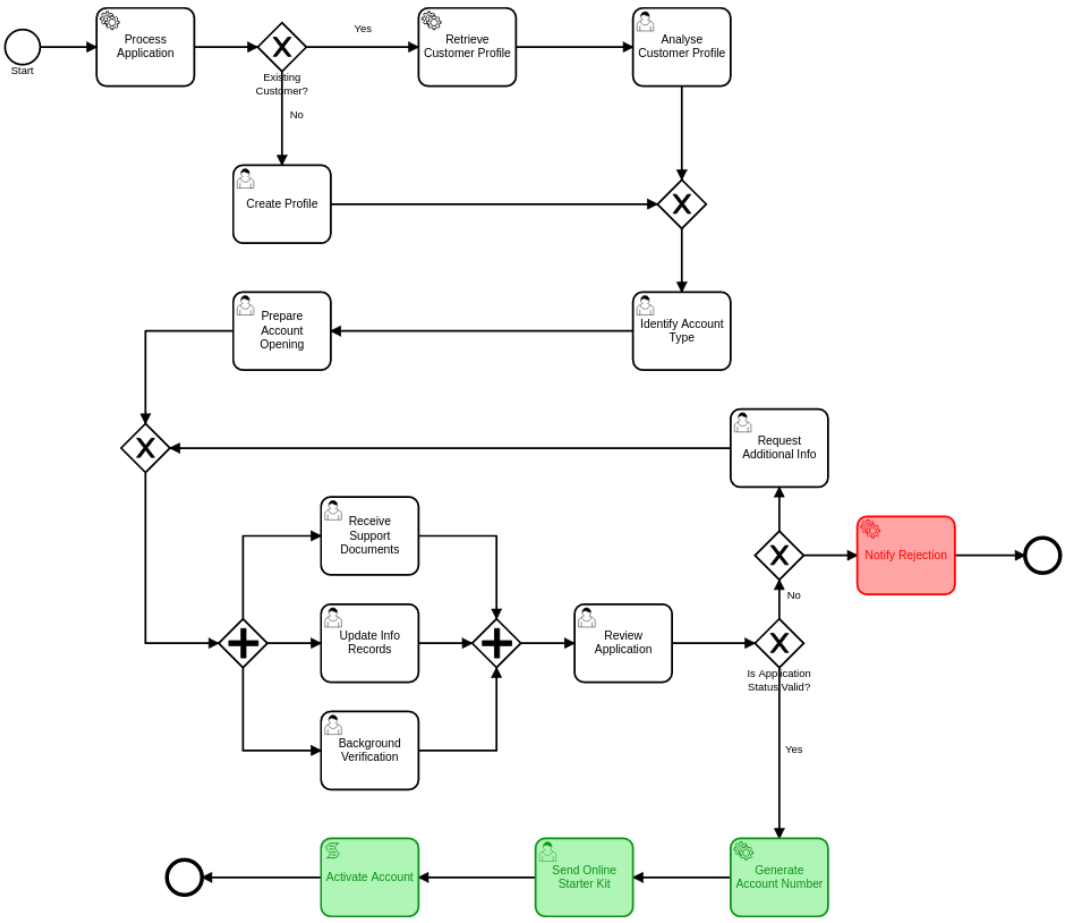

Figure 3.3: BPMN Process of Figure 2.2 directly colored

We can see that the task *Notify Rejection*, that violates the property, has been colored in red, while the task *Generate Account Number* and its descendants *Send Online Starter Kit* and *Activate Account*, that satisfy the property, have been colored in green.
Note that, in the rest of this work, we may refer to *semantically correct coloration*. These terms mean that, for any green (resp. red) node E of the BPMN process, there does not exist any path $p = (e_1, ..., e_n) \in Out(E)$ such that $\exists i \in [1...n]$ s.t. e_i is colored in red (resp. green). For the direct coloration process, the semantically correct coloration is ensured by the matching analysis phase.

3.2 Indirect Coloration

In the previous section, we described the approach when a matching was found between the CLTS and the initial BPMN process. Unfortunately, this is not always the case. In some cases, no matching can be found between the initial BPMN and the CLTS. In such cases, we propose another solution. This solution consists in transforming the CLTS into a new BPMN process, syntactically different from the initial one, but semantically equivalent. This transformation is performed by Algorithm 2.

The idea of this transformation is the following: we traverse the whole CLTS, and, for each state of the CLTS, we verify if it has already been visited. If this is the case, we retrieve the BPMN node corresponding to this state. Otherwise, we check the number of outgoing transitions of this state. If it is 0, the current state is a sink state, so we generate a new BPMN end event, and associate it to this state. If it is 1, we generate a new BPMN task, and associate it to this state. Otherwise, we generate a new BPMN exclusive split gateway, associate it to this state, and generate a new BPMN task for each of its outgoing transitions, that we link to the generated gateway. This transformation is illustrated in Appendix A.2.

We have seen during the introduction of VBPMN that the transformation of a BPMN process into a CLTS preserves its semantics. Here, the transformation performed preserves the semantics of the CLTS. By transitivity, the semantics of the initial BPMN process is preserved, and the generated BPMN process is semantically equivalent to the initial one. Once generated, the new BPMN process is colored according to the CLTS, by following the same procedure as in the direct coloration step. Finally, we give the generated BPMN process back to the user. We called this solution *Unfolding*. This is a convenient solution, having nonetheless the drawback of possibly generating large BPMN processes with high number of nodes. To overcome this, we propose an extent to this solution, in which we try to identify some patterns in the unfolded BPMN, and see whether they are semantically equivalent to unfolded parallel gateways. If it is the case, those parts of the BPMN process are folded, which can lead to a significant reduction of the size (in terms of number of nodes) of the BPMN process. We called this extent *Folding*. The goal of this extent is to provide a better readability and understandability for the user.

Algorithm 2 CLTS to BPMN Algorithm

```
1: procedure BUILDGRAPH(CurrentNode, CurrentState, StateNodeCorrespondences)
2:     if CurrentState = null then return
3:     NumberOfChildren ← CurrentState.numberOfOutgoingTransitions()
4:     StateAlreadyVisited ← StateNodeCorrespondences.contains(CurrentState)
5:
6:     if NumberOfChildren = 0 then
7:         if StateAlreadyVisited then
8:             NextNode ← StateNodeCorrespondences.get(CurrentState)
9:         else
10:            if CurrentState is green then
11:                NextNode ← GENERATENEWGREENENDEVENT()
12:            else
13:                NextNode ← GENERATENEWREDENDEVENT()
14:            end if
15:            StateNodeCorrespondences.put(CurrentState,NextNode)
16:        end if
17:        LINK(CurrentNode,NextNode)
18:    else if NumberOfChildren = 1 then
19:        if StateAlreadyVisited then
20:            NextNode ← StateNodeCorrespondences.get(CurrentState)
21:        else
22:            NextNode ← GENERATENEWTASK(CurrentState.uniqueOutgoingTransition)
23:            StateNodeCorrespondences.put(CurrentState,NextNode)
24:            BUILDGRAPH(NextNode,CurrentState.uniqueOutgoingTransition.nextState)
25:        end if
26:        LINK(CurrentNode,NextNode)
27:    else
28:        if StateAlreadyVisited then
29:            NextNode ← StateNodeCorrespondences.get(CurrentState)
30:        else
31:            NextNode ← GENERATENEWEXCLUSIVEGATEWAY()
32:            StateNodeCorrespondences.put(CurrentState,NextNode)
33:            for outgoingTransition of CurrentState do
34:                ChildNode ← GENERATENEWTASK(outgoingTransition)
35:                LINK(NextNode,ChildNode)
36:                BUILDGRAPH(ChildNode,outgoingTransition.nextState)
37:            end for
38:        end if
39:        LINK(CurrentNode,NextNode)
40:    end if
41: end procedure
```

3.2.1 Unfolding Only

From the matching analysis phase, we exhibited 2 general conditions in which direct coloration can not be performed: either **(a)** we detected transitions with the same label having different colors, or **(b)** at least one of the faulty states of the CLTS could not be mapped to any node of the initial BPMN process. These cases were found by reasoning on the CLTS, which is the semantic model of the BPMN process. One can name these 2 cases *semantic cases*. On the other side, by reasoning on the structure and the syntax of the BPMN process, and according to the subset of the BPMN syntax managed in this work, we were able to find 4 cases in which direct coloration could not be performed. We named them *syntactic cases*. In the rest of this work, we mostly use the syntactic cases, as they offer a visual representation and a better comprehension of the problems that they involve. Note that these 4 cases have a common point, which is that the counterexample(s) returned by the model checker can not be represented on the initial BPMN process, due to syntactic differences with its corresponding CLTS.
These 4 cases are, namely:

(i) Parallel Gateway Expansion

(ii) Inclusive Gateway Expansion

(iii) Loop Unrolling

(iv) Property Violation Before Merge Gateway

Now that we have identified those 4 cases, let us detail them.

Parallel Gateway Expansion

In this case, the difference comes from the transformation of the BPMN process into an LTS. Indeed, we mentioned earlier that the LTS representation does not allow parallelism. To be transformed into LTS, the BPMN process needs a semantically equivalent rewriting of its parallel portions, into sequential ones. According to Milner's expansion theorem [4], one is able to transform parallelism into choice and sequence. This is what is done during this transformation. Figure 3.4 shows an example of this rewriting, where we replaced the LTS by its semantically equivalent BPMN version for a better understanding of the process. This rewriting can be qualified of *full rewriting* because the resulting gateway contains only exclusive gateways.

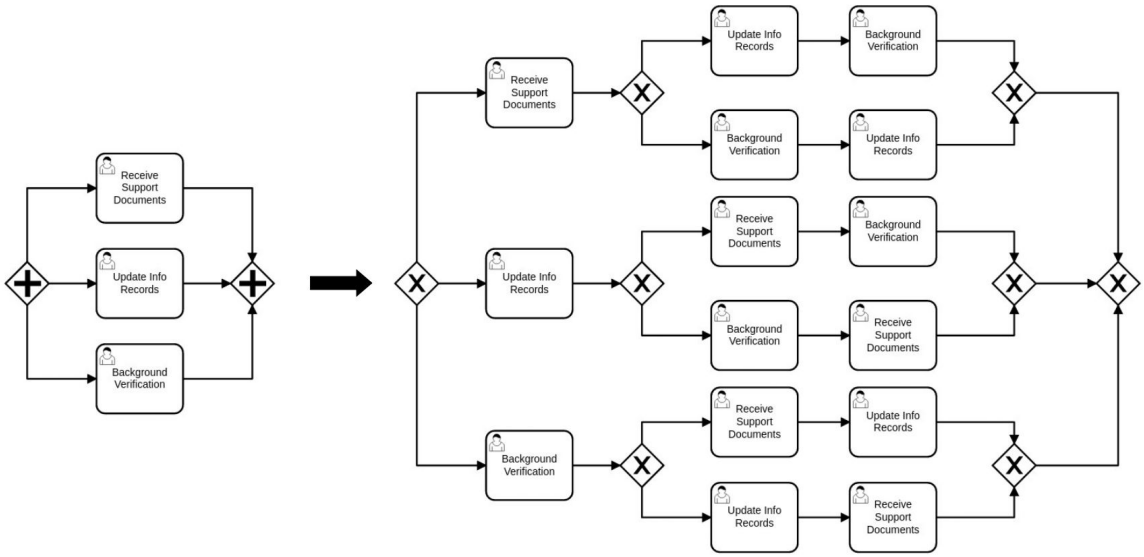

Figure 3.4: Rewriting of a BPMN parallel gateway using BPMN exclusive gateways only

As the reader can see, such a rewriting exposes all the possible inner paths of a parallel gateway. Each of these inner paths is representing an order in which the tasks belonging to the parallel gateway can be performed. Moreover, by considering this rewriting, one can easily find an MCL property satisfied by some paths and violated by some others. With such a property, some paths of the expanded parallel gateway would be red, while others would be green. Therefore, no semantically correct coloration of the initial parallel gateway can be performed, and consequently no coloration of the initial BPMN process.
The couples of BPMN processes and MCL properties belonging to this case can be formally characterized as follows: *let* $B = (S_N, S_E)$ *be a BPMN process and P an MCL property. The couple (B,P) belongs to this case if* $\exists g \in S_N$ *a parallel split gateway s.t.* $\exists p_1, p_2 \in Inner(g), p_1 \neq p_2$ *s.t.* p_1 *satisfies P and* p_2 *violates it.*

Inclusive Gateway Expansion

This case is similar to the previous one. Indeed, inclusive gateways are similar to parallel gateways, and are also involving parallelism, so they need to be rewritten. Figure 3.5 shows an example of this rewriting.

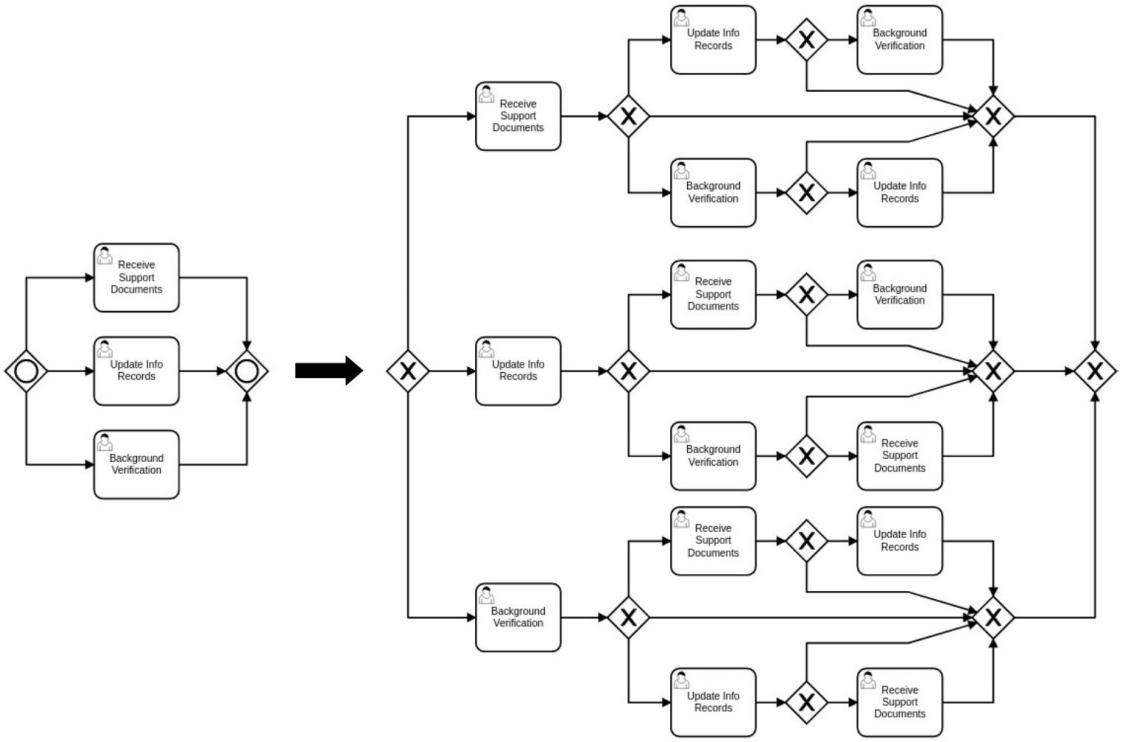

Figure 3.5: Rewriting of a BPMN inclusive gateway using BPMN exclusive gateways only

The problem encountered here is the same than the one for parallel gateways: if some paths of the rewritten inclusive gateway are satisfying the property, and others are violating it, no semantically correct coloration of the initial inclusive gateway is possible.

Loop Unrolling

In this case, the difference does not come from the transformation of the initial BPMN process into an LTS. In fact, here, the difference is induced by the MCL property. Indeed, some safety properties may concern a certain number of loop iterations $n \in \mathbb{N}$. In other words, it means that some tasks or sequence of tasks belonging to the property are repeated in it, once per loop iteration. In such cases, the counterexample(s) returned by the model checker will contain n times the sequence of actions belonging to the loop. These sequences of actions are called *unrolled loops*. The CLTS generated by CLEAR will then also contain n times this sequence of actions. Note that unrolling also happens if a task involved in the MCL property belongs to a loop, and there exists no path starting from the initial event and ending with an end event which contains this task. More formally, *let* $B = (S_N, S_E)$ *be a BPMN process and* $P = [true^* \,.\, "T_1" \,.\, true^* \,.\, "..." \,.\, true^* \,.\, "T_n" \,.\, true^*]$ *false an MCL property, where* $T_1, ..., T_n \in S_N$ *. If* $\forall i \in [1...n], \exists t_i \in P$ *s.t.* t_i *is in loop and* $\nexists$ *path* $p = (e_1, ..., e_n)$ *between the initial event and an end event s.t.* $t_i \in p$ *and* $\forall j,k \in [1...n], j \neq k \implies e_j \neq e_k$ *, then the couple (B, P) belongs to this case.*

To illustrate this case, we can take back our account opening example, along with the property *I do not want to ask for additional information more than twice*, which can be rewritten in MCL as: [*true* . Request Additional Info . true* . Request Additional Info . true* . Request Additional Info . true**] *false*.

A counterexample of this property is the sequence of actions containing 3 times the sequence of actions belonging to the loop.

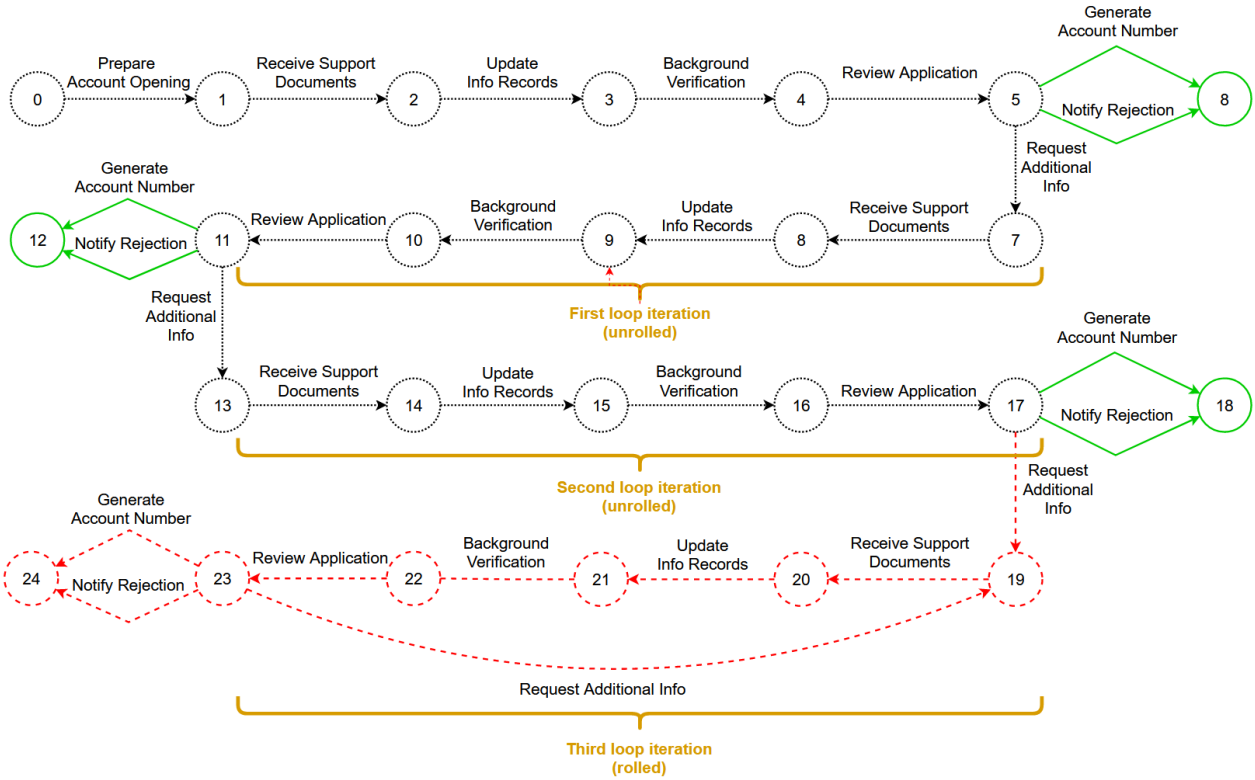

Figure 3.6: Loop unrolling (CLTS)

As the reader can see on Figure 3.6, the CLTS shows the 2 unrolled versions of the loop in black, because they do not violate nor satisfy the property, and the rolled version of the loop in red, because it violates the property.
As some transitions have the same label but different colors, like for example the transition *Background Verification*, direct coloration can not be performed.

Property Violation before Merge Gateway

In this case, the difference is caused by the shape of the BPMN process. It happens in BPMN processes containing balanced exclusive gateways, if the MCL property to verify can either be satisfied or violated by taking one or another inner path of a balanced exclusive gateway. As a reminder, inner paths are paths starting from a split gateway and ending with the corresponding merge gateway, if it exists. This case can be formalized as *let* $B = (S_N, S_E)$ *be a BPMN process and P an MCL property. The couple (B,P) belongs to this case if* $\exists g \in S_N$ *a balanced exclusive split gateway for which* $\exists p_1, p_2 \in Inner(g)$ *s.t.* p_1 *violates P and* p_2 *satisfies P.*
Finally, to illustrate **(iv)**, we can take again our account opening example along with the property *I do not want new customers*, which can be translated in MCL as: [*true** . *Create Profile* . *true**] *false*. The counterexample here is almost trivial: any path going through the task *Create Profile* does not satisfy the property, while any path going through the task *Retrieve Customer*

Profile satisfies it. In this case, one could be tempted to propose the coloration shown in Figure 3.7.

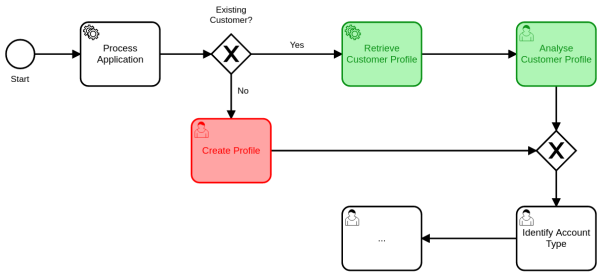

Figure 3.7: Semantically incorrect coloration of a BPMN process

Nonetheless, this coloration is semantically incorrect, because tasks *Create Profile* and *Analyse Customer Profile*, that are colored differently, lead to the same exclusive gateway. In fact, there is no semantically correct way of coloring the task *Identify Account Type*, nor the next ones, as they are part of both correct and incorrect paths. In such case, the CLTS will contain two "versions" of the reachable tasks starting from task *Process Application*. A correct version which contains the task *Retrieve Customer Profile*, and an incorrect one containing the task *Create Profile*. This will lead to transitions with the same label colored differently, forbidding direct coloration.

Summary

For now in this chapter, we have detailed cases where the initial BPMN process and the CLTS had a matching, and 4 cases where the initial BPMN process and the CLTS could not have a matching. Considering these 4 cases, one may wonder about other possible cases that may prevent a matching to be found. We claim that, except those 4 cases, or any combinations of them, a matching necessarily exists between the initial BPMN process and the CLTS. In other terms, in this approach, we propose a *full covering* of any kind of couple (BPMN process, MCL property).

Proposition 1 (Full Covering)
Let $B = (S_N, S_E)$ *be a BPMN process, P an MCL property, and M the CLTS generated from (B,P).*
$(((B,P) \notin$ ***Case (i)***$) \wedge ((B,P) \notin$ ***Case (ii)***$) \wedge ((B,P) \notin$ ***Case (iii)***$) \wedge ((B,P) \notin$ ***Case (iv)***$))) \implies$ *Match(B,M)*

Proof (Sketch). A couple (B,P) that does not belong to any of the presented cases is a couple for which **((i))** the corresponding CLTS *M* does not contain unrolled loops, **(ii)** P is not satisfied nor violated before a merge gateway, and **(iii)** all the parallel and inclusive gateways are either fully satisfying, fully violating or neutral regarding the property.

Condition 1: No transition of the CLTS having the same label is colored differently

(i) & **(ii)** $\implies$ **(1)** No transition of the initial BPMN process is duplicated in the CLTS.

(iii) $\Longrightarrow$ **(2)** All the transitions belonging to parallel or inclusive gateways have the same color.
(1) & **(2)** $\Longrightarrow$ There is no couple of transitions (t_1, t_2) in the CLTS such that $t_1 \neq t_2, label(t_1) = label(t_2)$ and $color(t_1) \neq color(t_2)$.
Thus, Condition 1 is satisfied.

Condition 2: Each faulty state of the CLTS must be linked to a unique BPMN node

We have seen earlier in the report that the initial BPMN process is semantically equivalent to the CLTS. To achieve this, each node of the BPMN process needs to have its equivalent representation in the CLTS. Then:

(i) & **(ii)** & **(iii)** $\Longrightarrow$ **(3)** All the states of the CLTS have a corresponding node in the initial BPMN process, except the ones belonging to unfolded parallel or inclusive gateways.

But **(iii)** $\Longrightarrow$ **(4)** All the states belonging to an unfolded parallel or inclusive gateway are either green, red, or neutral, so they can not be faulty states.

Then, **(3)** & **(4)** $\Longrightarrow$ **(5)** Each faulty state of the CLTS has a corresponding node in the initial BPMN process.
Thus, Condition 2 is satisfied.
Finally Condition 1 & Condition 2 $\Longrightarrow$ Match(B,M).

To conclude this section, as indicated by its name, the unfolding phase outputs an unfolded version of the initial BPMN process. This means that the number of nodes composing the generated BPMN process is greater or equal than the number of nodes composing the initial BPMN process. As one of the main goals of this approach is to return to the user a BPMN process as small as possible in number of nodes, we may want to reduce the size of this unfolded BPMN process when possible. The solution that we propose here to perform this reduction is called *folding*. The folding can be applied to unfolded parallel and inclusive gateways (cases **(i)** & **(ii)**). In particular, in this work, we focused on parallel gateways (case **(i)**) having inner paths of size 1 and no loop. Handling more generic parallel gateways is part of future work, as well as managing inclusive gateways (case **(ii)**). We also propose an idea to improve the management of unrolled loops (case **(iii)**) and another one to avoid the merge gateway problem (case **(iv)**).

3.2.2 Unfolding & Folding

In this part, only exclusive gateways are considered, as inclusive and parallel gateways have been rewritten in their exclusive versions (presented in Figure 3.4 & 3.5). In this context, the word *gateway* will abusively designate exclusive gateways all along this subsection. Second, the term *folding* will be used to designate the transformation of a set of tasks belonging to an exclusive gateway to a set of tasks belonging to a parallel gateway. Finally, the word *BPMN process* will refer to the unfolded version of the initial BPMN process.

Before detailing the folding approach, we need to differentiate two types of gateways: the *nested gateways* and the *outer gateways*. The difference between them is that a nested gateway is included in at least 1 other gateway, while an outer gateway is not included in any other

gateway. Note that this a strict categorization, because any gateway is either a nested gateway, or an outer gateway.

Definition 13. *(Nested Gateway) Let $B = (S_N, S_E)$ be a BPMN process, and $S_{G_N} \subset S_N$ the set of gateways composing B. $\forall g \in S_{G_N}$, g is a nested gateway if and only if $\exists$ path $p = (e_1, ..., e_n)$ s.t. $e_n = g$ and $\exists i \in [1...(n-1)]$ s.t. e_i is a split gateway and $\exists p' = (e'_1, ..., e'_m) \in Inner(e_i)$ s.t. $\exists j \in [1...m]$ s.t. $e_j = g$.*

Definition 14. *(Outer Gateway) Let $B = (S_N, S_E)$ be a BPMN process, and $S_{G_N} \subset S_N$ the set of gateways composing B. $\forall g \in S_{G_N}$, g is an outer gateway if and only if $\nexists$ path $p = (e_1, ..., e_n)$ s.t. $e_n = g$ and $\exists i \in [1...(n-1)]$ s.t. e_i is a split gateway and $\exists p' = (e'_1, ..., e'_m) \in Inner(e_i)$ s.t. $\exists j \in [1...m]$ s.t. $e_j = g$.*

The folding approach proposed is composed of 4 main steps:

- **Step 1**, in which we compute all the outer gateways of the BPMN process that contain only nodes with identical colors, and separate each outer gateway and its subnodes into groups.
- **Step 2**, in which we compute information about each gateway of each group. From these information, we initialize some data struture that we call *metadata*.
- **Step 3**, in which we fill the metadata of each gateway of each group, in order to know if it is reducible, and, if yes, how.
- **Step 4**, in which we build the most reduced version of each gateway group with the help of the metadata, and replace it by the generated one in the BPMN process.

Step 1 – Grouping

The main goal of this step is to separate the nodes of the BPMN process having the same color into groups. The specificity of each group is that it contains exactly 1 outer gateway, and an undefined number of nested gateways. The algorithm used to generate those groups is the following: starting from each first colored node, the algorithm analyses all the outgoing paths to find a split gateway. If this split gateway is balanced, it retrieves its corresponding merge gateway, and a group is created containing all the nodes belonging to this gateway. Then the algorithm continues its execution on each child of the merge gateway, looking for other split gateways. If the split gateway is non-balanced, a group is generated, containing all the nodes reachable from this split gateway, and the research on the current path ends. As this algorithm traverses the whole graph, we are sure not to miss any gateway.

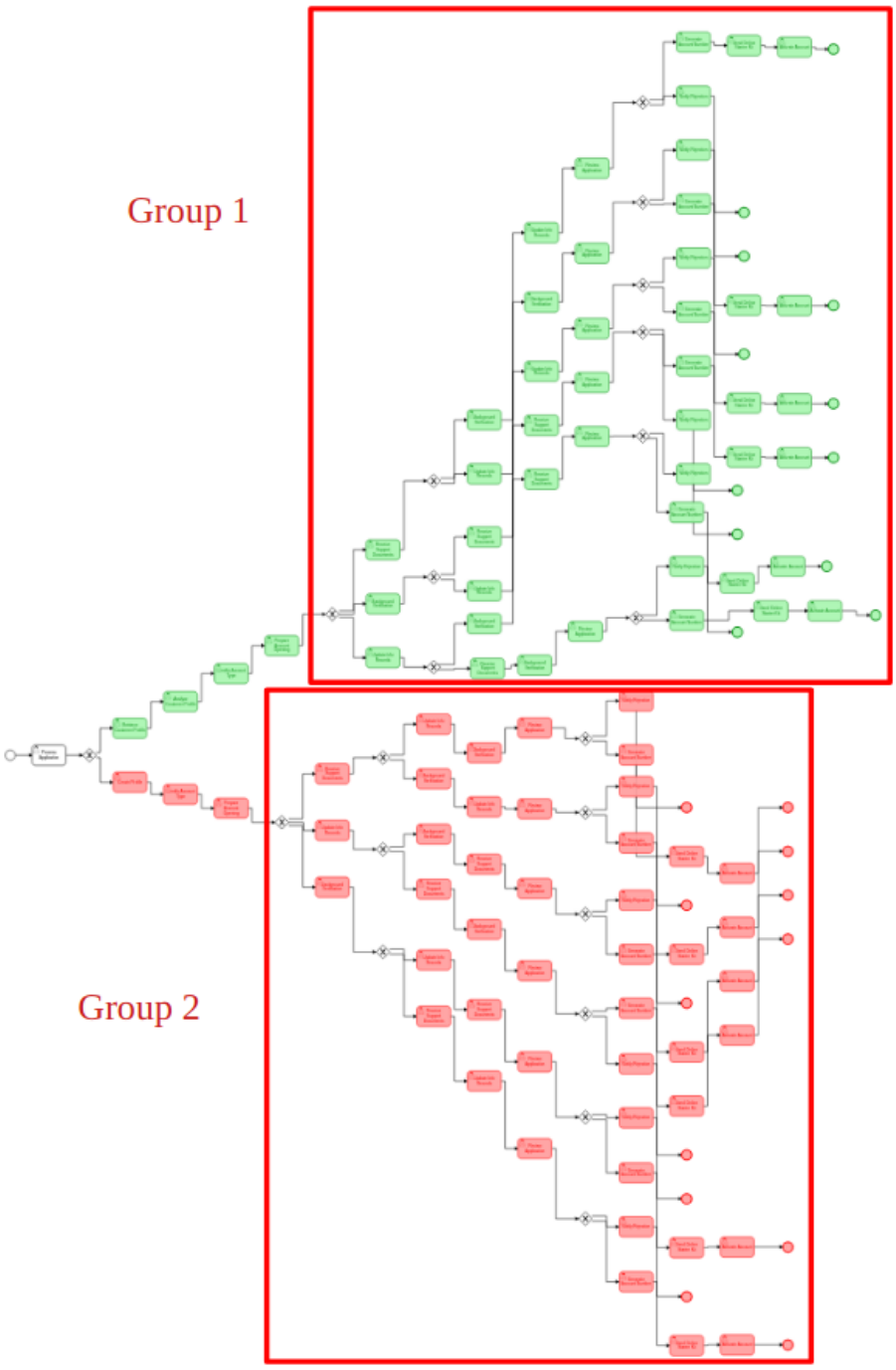

Figure 3.8: Grouping detected 2 groups

Figure 3.8 illustrates the result of the grouping on our account opening example and the property [*true** . *Create Profile* . *true**] *false*. As the reader can see, 2 groups have been detected: 1 in the green part of the BPMN process, and 1 in the red part. In this example, there is no merge gateway, so the first split gateway encountered in the red part and in the green part are identified as outer gateways, generating 2 groups.

Step 2 – Initialization

Now that we have defined our groups, we will compute some information for each gateway inside the current group. These information are called *metadata*. The initialization consists in attributing to each gateway inside the current group a positive integer value representing the foldability that it can not exceed. We called this value *maximum foldability*. What we mean here is that a gateway having a maximum foldability of 3 can ideally be replaced in this approach by a parallel gateway of size 3.

Note that we used the word *ideal* to describe the notion of maximum foldability. Indeed, the size of the parallel gateway replacing the current exclusive at the end of this procedure can be lower than this value. It can even happen that this gateway is, in fact, not foldable, and will not be replaced by a parallel gateway at the end of this procedure.
In our case, the maximum foldability of a gateway is initialized to its size. To set this value, we traverse the current group and create a new metadata for each new gateway encountered.

Step 3 – Foldabilities Computation

Now that we have initialized each gateway of each group, we can compute their foldabilities. This phase is separated in two parts. The first part concerns gateways having a maximum foldability of 2, for which we analyze the outgoing paths to fill the metadata. The second part is dedicated to gateways having a maximum foldability strictly higher than 2, for which we analyze the metadata of their nested gateways, if they exist. This step is the most crucial one, as it determines if and how each gateway can be folded. To do so, we perform an analysis of each gateway's metadata, starting from the gateways of smallest maximum foldability, *e.g.* 2.
Step 3.1 describes the management of gateways having a maximum foldability of 2 while **Step 3.2** describes the management of gateways having a maximum foldability strictly higher than 2.

Step 3.1 – Maximum foldability of 2

The technique used in this step can be somehow related to the process mining technique [32], in which we analyze the execution traces of a program to exhibit its behaviour.
Indeed, in this step, we analyse all the paths starting from the current gateway in order to extract 3 information about it. The first two concern the foldability of the gateway. To decide whether a gateway is foldable or not, we analyze its outgoing path to check if there exist a couple of paths for which the first node of the first path corresponds to the second node of the second path, and vice versa. Such paths are called *size-2 diamond-shaped paths*.

Definition 15. *(Size-2 Diamond-Shaped Paths) Let G_E be an exclusive split gateway and $Out(G_E)$ the set of outgoing paths of G_E.*
$\forall (i,j) \in [1;|Out(G_E)|], i \neq j$, if $\exists p_i = (e_{i,1},...,e_{i,n}), p_j = (e_{j,1},...,e_{j,m}) \in Out(G_E), n,m \geq 2$ s.t. $e_{i,1} = e_{j,2}$ and $e_{i,2} = e_{j,1}$, then p_i and p_j are size-2 diamond-shaped paths.

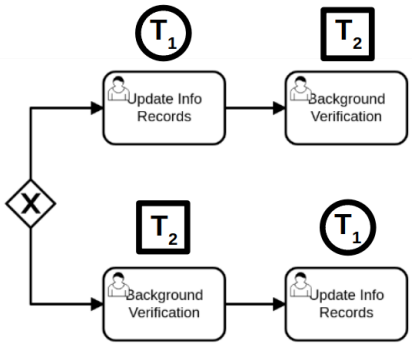

Figure 3.9: Example of gateway having size-2 diamond-shaped paths

Figure 3.9 illustrates a gateway having size-2 diamond-shaped paths. As the reader can see, the first node of the first path corresponds to the second node of the second path, and vice versa. If such paths are found, we compute the second information. To compute it, we differenciate the outgoing paths of the current gateway in 3 sets: those starting with the first size-2 diamond-shaped path, those starting with the second size-2 diamond-shaped path, and the others that are ignored. We keep the first 2 sets, and remove the first 2 tasks from each path of each set. The remaining paths in each sets are called *out-of-scope paths*.

Definition 16. *(Out-of-scope Paths) Let G_E be an exclusive split gateway of maximum foldability 2, (p_1, p_2) the size-2 diamond-shaped paths of G_E, and DSP the set of outgoing paths of G_E starting with p_1 or p_2. $\forall p = (e_1, ..., e_n) \in DSP$, if size(p) ≥ 3, then the path $p' = (e_3, ..., e_n)$ is an out-of-scope path.*

Figure 3.10 shows an example of 4 out-of-scope paths belonging to 2 distincts sets of out-of-scope paths. As the reader can see, rounded tasks and squared tasks form two distinct out-of-scope paths, belonging to the same set. The same remark applies to triangled and diamonded tasks.

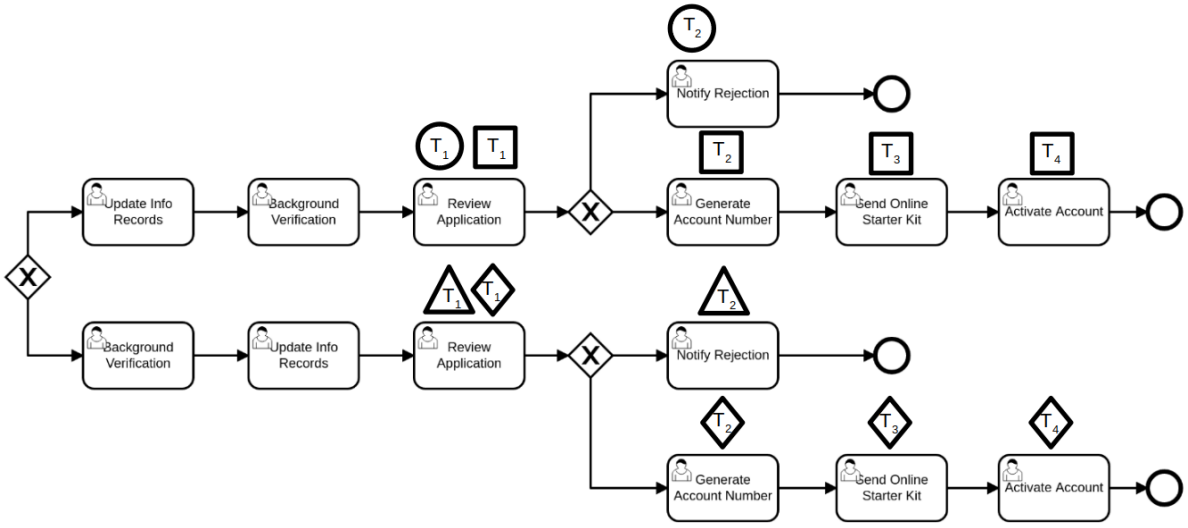

Figure 3.10: Example of gateway having 2 sets of 2 out-of-scope paths

Once we have computed those 2 sets of out-of-scope paths, we compare them. If they are identical, we conclude that the current gateway is foldable, with an effective foldability value of 2. We then fill its metadata by marking the gateway as foldable in a parallel gateway of size 2. The tasks belonging to this parallel gateway corresponds to the first two tasks of the size-2 diamond-shaped paths found. We also store all the out-of-scope paths computed.
If the gateway has been marked as foldable, we compute the third information. This information is used to know whether all the children of the current gateway are parallelizable or not. In practice, if the number of children of a gateway exceeds its effective foldability value, then we know that some of these children are not parallelizable. We called this information *gateway purity*.

Definition 17. *(Gateway Purity) Let G_{FE} be a foldable exclusive split gateway of effective foldability $n \in \mathbb{N}_{\geq 2}$. G_{FE} is pure if and only if size$(G_{FE}) = n$.*

Once the purity characteristic has been evaluated, we fill the metadata with the paths making the gateway not pure, if they exist. We call them *impure paths*. Figure 3.11 shows an example of impure path. In this example, the path containing the tasks *Inspect Documents* and *Perform Data Analysis* is an impure path. Indeed, this gateway is reducible in a parallel gateway of size 2 containing the tasks *Update Info Records* and *Background Verification*, so the path circled on the figure is an impure path.

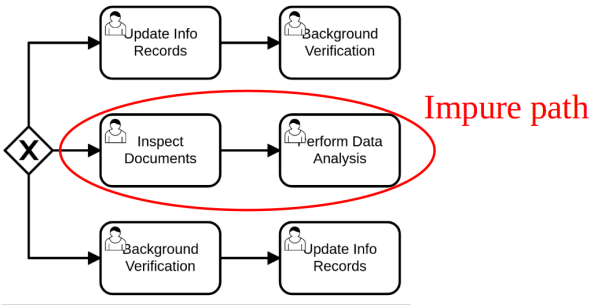

Figure 3.11: Example of gateway having 1 impure path

Now that we have computed all the information concerning gateways of maximum foldability 2, we can go to Step 3.2 and compute the information of gateways with a maximum foldability of 3 or more.

Step 3.2 – Maximum foldability strictly higher than 2

In this step, we manage the gateways of maximum foldability strictly higher than 2. We consider them in increasing maximum foldability order, meaning that we first consider gateways of maximum foldability 3, then 4, and so on. The idea here is to build the metadata of the current gateway with the help of the metadata of its nested gateways having a maximum foldability value lower by 1 compared to it. For example, the metadata of a gateway of maximum foldability 3 is built with the help of the metadata of its nested gateways of maximum foldability 2, if they exist.

Now let us remind the parallel gateway rewriting presented in Figure 3.4. We qualified it of *full*

rewriting, in the sense that the rewritten gateway only contains exclusive gateways. Nonetheless, this rewriting is not the only semantically equivalent rewriting possible. Another one, which can be qualified of *partial rewriting*, is proposed in Figure 3.12. In this version, the size-2 diamond-shaped paths visible in Figure 3.4 have been kept parallel. The purpose of this partial rewriting example is to give you an insight of the working of the algorithm proposed in this step.

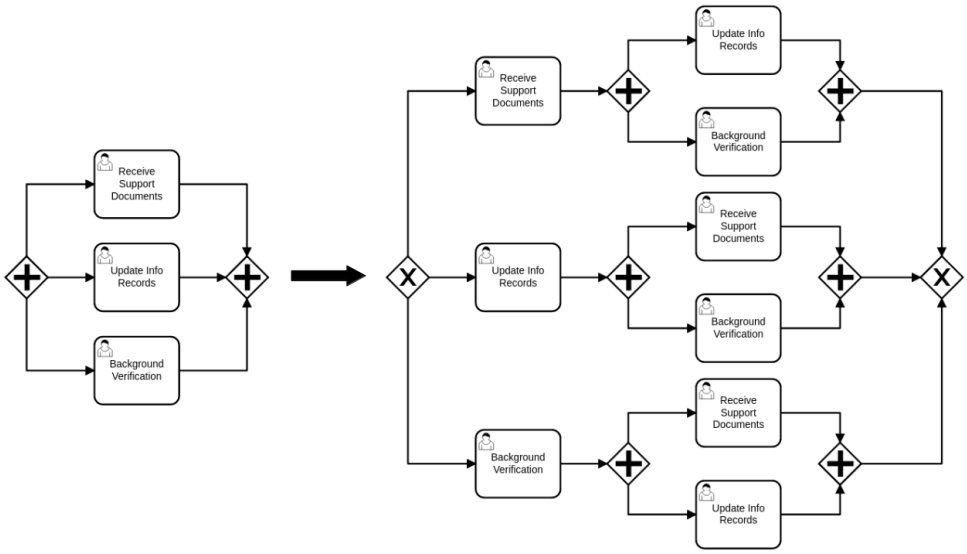

Figure 3.12: Example of size 3 gateway partially rewritten

Indeed, in this figure, the parallel gateways appearing in the partially rewritten gateway (right) represent the knowledge we learned from the metadata of their exclusive version. This example can be derived for any gateway of maximum foldability higher than 2. From this example, we can extract the definition of *foldable gateways* detected by this approach.

Definition 18. *(Foldable Gateway) Let G_E be an exclusive split gateway of size $n \in \mathbb{N}_{\geq 3}$.*
G_E is a foldable gateway if and only if, $\forall i \in [1...n], \forall c_i \in Child(G_E), c_i$ is a BPMN task, $size(c_i) = 1$, $c_{c_{i,1}} \in Child(c_i)$ is a pure parallel split gateway of size $n-1$ without any out-of-scope path, $\forall p \in Inner(c_{c_{i,1}})$, $size(p) = 1$ and $Child(c_{c_{i,1}}) = Child(G_E) \setminus c_i$.

For a better understanding of this definition, we propose to illustrate it visually in Figure 3.13. This figure represents a gateway of size n that is conformant to Definition 18, and consequently foldable in a parallel gateway of size n containing tasks $(T_1, ..., T_n)$. As the reader can see, the exclusive gateway is of size n. Each of its children is a task having exactly 1 child. Each child of a task is a pure parallel split gateway of size $n-1$ without any out-of-scope path. Each path of a pure parallel split gateway is of size 1, and the set of children of the pure parallel gateway considered is exactly the set of children of the initial exclusive gateway deprived of the task initiating the current path. For example, for the path starting with task T_i, the parallel gateway contains the set of tasks $\{T_1, ..., T_n\} \setminus T_i$. In these conditions, the initial exclusive gateway can indeed be folded in a parallel gateway of size n containing tasks $(T_1, ..., T_n)$.

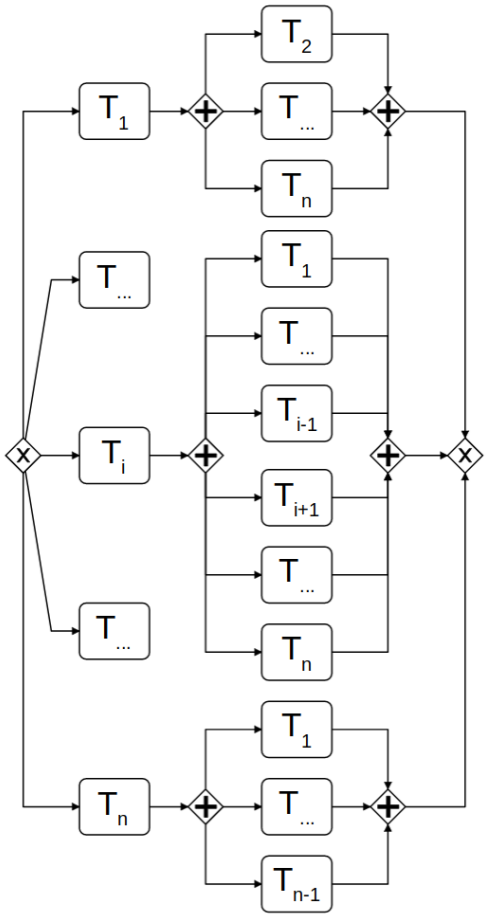

Figure 3.13: Size n foldable gateway

This computation is performed by Algorithm 3, which makes use of Definition 18 to fulfill the metadata of gateways of maximum reducibility $n > 2$.
The algorithm can be decomposed in 3 main steps:

(i) It checks each child of the current gateway. If the child is a task and has a single child which is a pure exclusive split gateway of size $n-1$ with a maximum foldability of $n-1$, it is kept. Otherwise, the child is discarded. This step aims at reducing the computational cost of the next step.

(ii) Among the children not discarded in the previous step, it computes all the combinations of children of size n.

(iii) It compares each combination found to Definition 18. If a combination matches the definition, it is used as pivot to fill the metadata of the current gateway. Note that several combinations may be valid regarding the definition. In such case, the first valid one will be kept, and others will be discarded.

If a combination is found, the metadata of the current gateway is filled to store its parallelizable tasks, its out-of-scope paths (if eligible) and its purity. Otherwise, we reduce by 1 the maximum

Algorithm 3 Compute Metadata of Gateways of Maximum Reducibility > 3

```
1: procedure COMPUTEMETADATA
2:     MaxFoldability = currentGateway.maxFoldability
3:
4:     while MaxFoldability ≠ 2 do
5:         PossibleChild ←DISCARDUNELIGIBLECHILD(currentGateway.child)
6:         ChildCombinations ←GETALLCOMBINATIONS(currentGateway.child, maxFoldability)
7:         ValidCombination ← False
8:
9:         for all Combination ∈ ChildCombinations do
10:            if Valid = False then Valid ← VALIDATECOMBINATION(Combination)
11:        end for
12:
13:        if Valid = False then MaxFoldability ← MaxFoldability − 1
14:    end while
15:
16:    function VALIDATECOMBINATION(Combination)
17:        OutOfScopePaths ← ∅
18:
19:        for all Element ∈ Combination do
20:            PresumedGatewayChild ← Element.firstChild
21:            GatewayMetadata ← GETMETADATAOF(PresumedGatewayChild)
22:
23:            if GatewayMetadata = null then return False
24:
25:            for all OtherElement ∈ Combination \ Element do
26:                if OutOfScopePaths = ∅ then
27:                    OutOfScopePaths ← GatewayMetadata.outOfScopePaths
28:                else
29:                    if OutOfScopePaths ≠ GatewayMetadata.outOfScopePaths then
30:                        return False
31:                    end if
32:                end if
33:
34:                OtherElementValid ← False
35:
36:                for ParallelElement ∈ GatewayMetadata.parallelElements do
37:                    if ParallelElement = OtherElement then OtherElementValid ← True
38:                end for
39:                if OtherElementValid = False then return False
40:            end for
41:        end for
42:        return True
43:    end function
44: end procedure
```

foldability of the current gateway, and, if the new maximum foldability is still strictly higher than 2, we redo Step 3.2, else, we do Step 3.1.

For a gateway of size n and maximum reduciblity $m \leq n$, we can compute the worst-case time complexity of the proposed algorithm as follows. Step 3.2 is repeated at most $m-2$ times and Step 3.1 one time.

Step 3.1, that has not been algorithmically detailed here, has a worst-case time complexity of $O(n^3)$.
The overall complexity of Step 3.2 can be computed from the complexity of its 3 main functions: *DiscardUneligibleChild*, *GetAllCombinations* and *ValidateCombination*. Function *DiscardUneligibleChild* has a complexity of $O(n)$, as it only traverses the set of children of size n once, while performing small verifications that runs in $O(1)$. In function *GetAllCombinations*, we compute all possible combinations of m among n in the worst case, which means $\binom{n}{m}$ computations. This gives us a worst-case time complexity of $O(n^m)$. Finally, for all $\binom{n}{m}$ combinations, we run function *ValidateCombination*, which performs in $O(m^3)$.

For Step 3.2, this gives us a global worst-case time complexity of $O(m) + O(n^m) \times m^3 = O(n^m) \times m^3$. As $n \geq m$, this complexity can be simplified to $O(n^m)$. Considering that Step 3.2 can be repeated at most $m-2$ times, we can compute the full complexity of this algorithm as $O(n^3) + (m-2) \times O(n^m)$. Knowing that $m \geq 3$, and for the same reasons as before, this complexity can be simplified to $O(n^m)$.

Of course, this time complexity is a worst-case theoritical complexity. In practice, the purpose of the function *DiscardUneligibleChild* is to reduce the number of child that are possibly foldable, by performing some verifications. This aims at reducing the number of combinations computed in function *GetAllCombinations*, while reducing by the same time its complexity. Moreover, it is very rare in practice to have parallel gateways of size 5 or more, which somehow bounds the complexity to $O(n^5)$. Finally, this remains a first version of the algorithm, which may be refined to achieve a better complexity.

Step 4 – Folding & Replacement

Now that the metadata of each exlusive gateway of each group has been computed, we can proceed to the folding of these exclusive gateways, and their replacement by their folded version. Here, by folding we mean generation of a folded version of the current gateway. In this step, we start from the gateways having the highest maximum reducibility value among all gateways. By doing so, we are sure to perform the most efficient folding possible, as no bigger foldable gateway including this gateway exists in the BPMN process. To generate the parallel gateway, we check the metadata of the current gateway. If the metadata says that the gateway is foldable, we put in parallel all its parallelizable tasks. Then, we add all the out-of-scope paths after the parallel merge gateway. Finally, if the gateway is not pure, we create an exclusive split gateway, having as outgoing paths all impure paths, plus the paths belonging to the generated parallel gateway. Once a gateway has been generated, it is directly added to the initial graph by replacing the initial exclusive gateway by the generated parallel gateway.

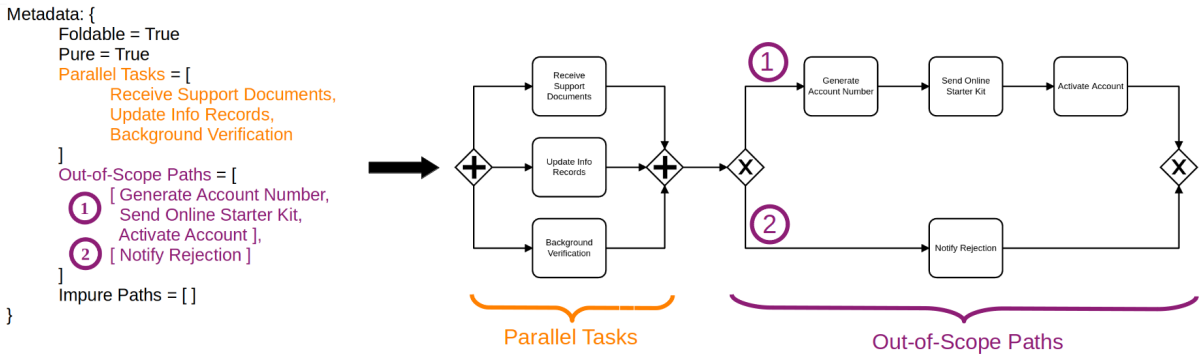

Figure 3.14: Generation of folded gateway from metadata

Figure 3.14 shows an example of this generation process from the metadata. On the left, we gave a textual representation of the information stored in the metadata. In this case, the reader can see that the gateway is foldable in a parallel gateway having 3 children (size 3), 2 out-of-scope paths, and no impure path. From these information, we are able to generate the folded gateway visible on the right side of the figure. As expected, we observe one parallel gateway containing the 3 parallelizable tasks, and two paths outcoming from this gateway: the out-of-scope paths.

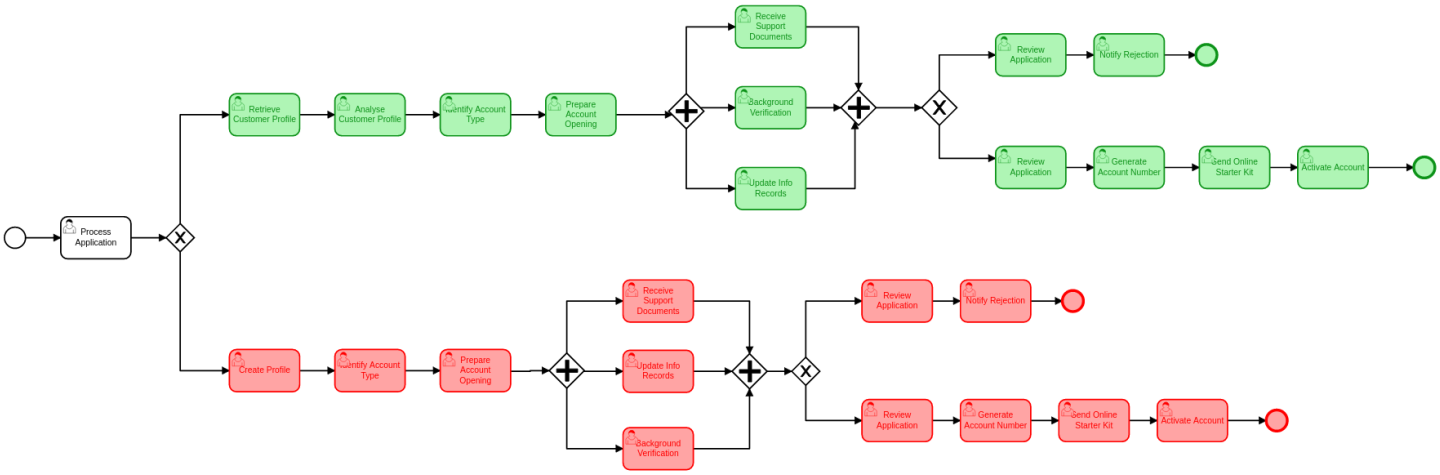

Figure 3.15: Folded version of Figure 3.8

While executing this algorithm on Figure 3.8, we are able to generate the BPMN process presented in Figure 3.15. One can observe that the folding is identical to the one proposed in Figure 3.14. We also performed the replacement step, in which each exclusive gateway is replaced by its semantically equivalent parallel gateway. Note that, as each step of this transformation preserves the semantics of the initial BPMN process (according to Milner's theorem [4]), the generated BPMN process is semantically equivalent to the initial one.

3.3 Other ideas

In the previous sections of this report, we described some approaches useful for coloring directly a BPMN process, and managing unfolded parallel and inclusive gateways. In this section, we present 2 ideas aiming at managing the 2 other problems, namely unrolled loops and merge gateways. Note that those ideas have not been fully explored yet, but doing it is part of future work.

3.3.1 Idea 1: Loop clustering

We called the first idea *loop clustering*. This idea aims at patching the problem shown in Figure 3.6, which concerns the unrolling of loops due to the MCL property. Indeed, if the MCL property concerns tasks belonging to a loop, and iterates them n times, then the loop will be unrolled n times in the BPMN process generated by the unfolding approach. Such behaviour has the drawback of possibly generating large BPMN processes, without providing a lot of information regarding the source of the violation.
To limit this behaviour, we decided to make use of other BPMN elements that have not been presented in this report: the group object and the text annotation, along with coloration. The interest of this idea is that, in any case, in the final graph, there would be at most 2 occurrences of the loop, without any unrolling. The first occurrence would be surrounded by a green cluster, with the annotation "$< n$ loop iterations", while the second occurrence would be surrounded by a red cluster, annotated "$\geq n$ loop iterations". Figure 3.16 illustrates this approach, based on the account opening process proposed in Figure 2.2 and the MCL property *I do not want to ask additional information more than 5 times*, translated to MCL as [*true** . *Request Additional Info* . *true** . *Request Additional Info* . *true** . *Request Additional Info* . *true** . *Request Additional Info* . *true** . *Request Additional Info* . *true**] *false*.

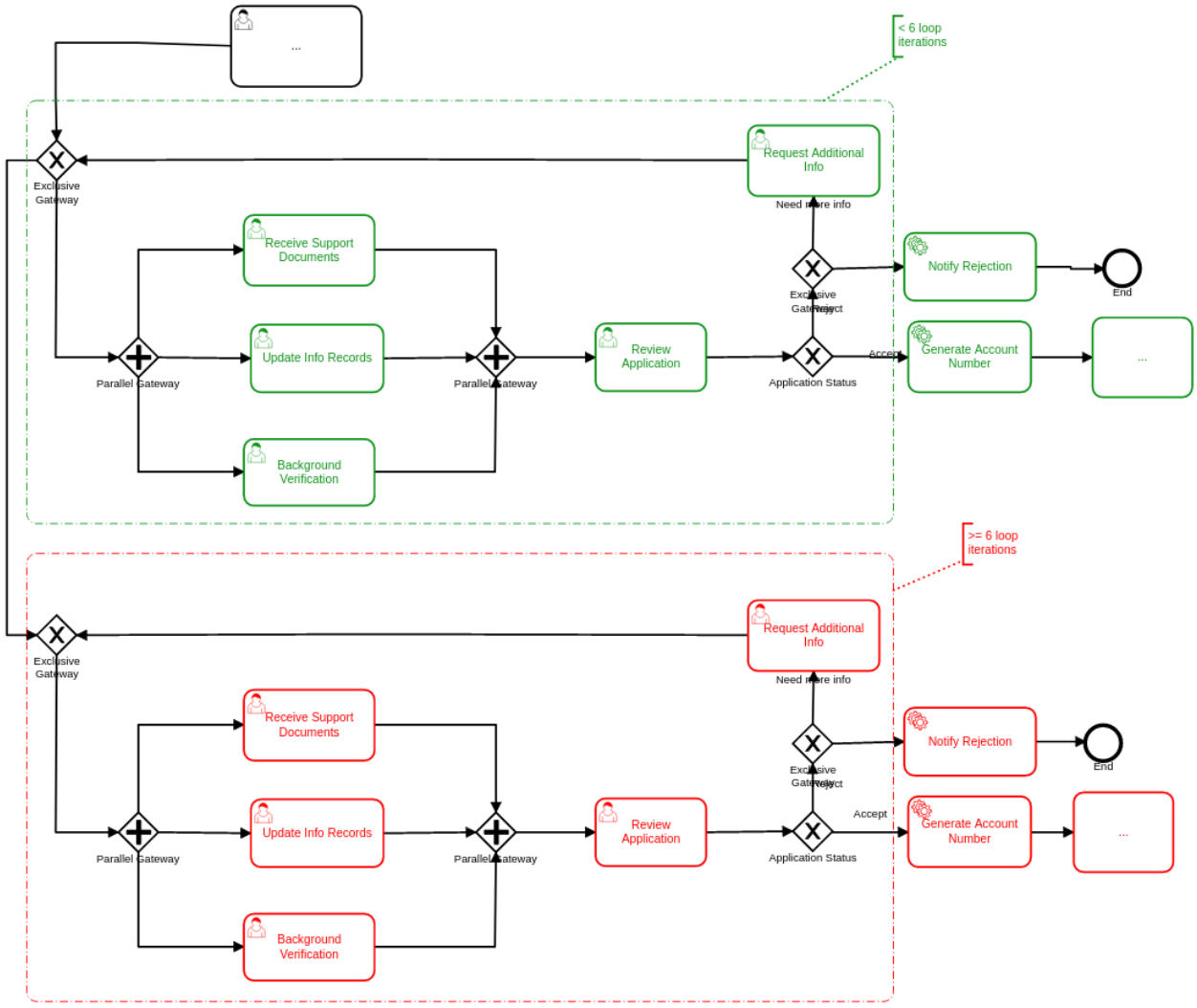

Figure 3.16: Loop clustering

As the reader can see, there is a first loop, colored in green, contained in a BPMN group with the annotation "< 6 loop iterations". This loop is followed by a second one, identical to the first one, but colored in red, and contained in group with the annotation "$\geq$ 6 iterations". In this case, we reduce the number of iterations of the loop from 6 to 2. By doing so, we remove 25 nodes from the generated BPMN process.

Nonetheless, one can see that we are able to find a path starting from a green node, and ending with a red node. Such paths should not exist in a semantically correct coloration. For this reason, this idea is still being deepened.

3.3.2 Idea 2: Color variations

We called the second idea *color variations*. This idea aims at performing coloration in a different way than the one proposed in this work, *i.e.* not only with green and red colors. The goal of this idea joins the common goal of this work: returning a BPMN process as syntactically close as possible to the initial one, while overcoming the merge gateway problem. Indeed, with this solution, one would be able to color directly the initial BPMN process, without finding a matching between the CLTS and the original BPMN process. This idea is based on the addition of a third color, yellow, being used to color a BPMN node that can either be satisfying or violating the property at some point. To keep all the information about the origin of the bug in the BPMN process, we color in yellow only nodes that do not correspond to a faulty state of the CLTS, and happen after a merge. Thereby, we give to the user a precise explaination of the bug, while avoiding the duplication of the subparts of the BPMN process in red and green. Figure 3.7 illustrates this idea on our account opening example and the property *I do not want new customers*, which can be translated in MCL as: [*true** . *Create Profile* . *true**] *false*.

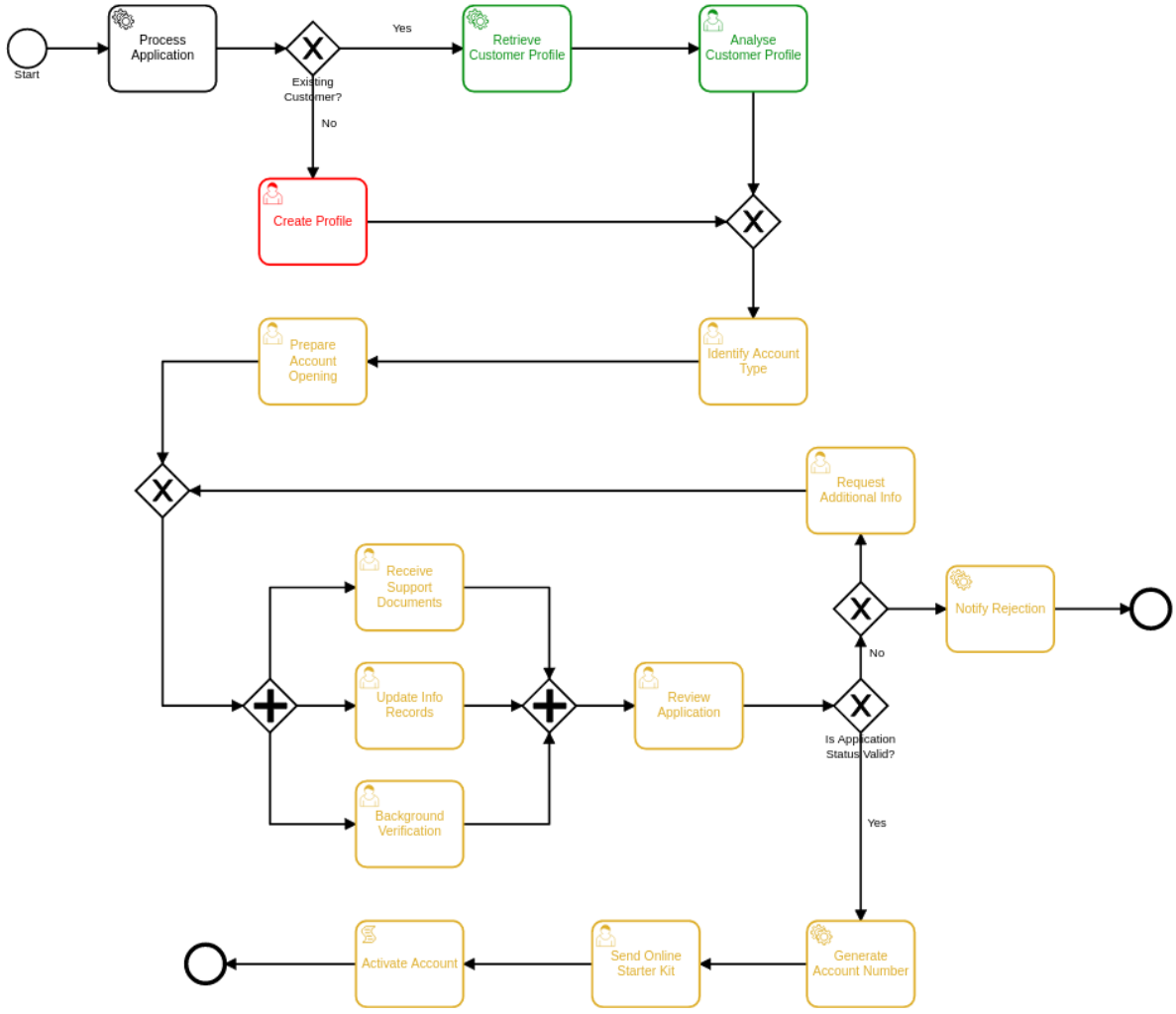

Figure 3.17: Example of color variations

One can see that the property is violated if we take the path containing the task *Create Profile*, while it is satisfied if we take the path containing the task *Retrieve Customer Profile*. Here, task *Identify Account Type* and the following ones do not provide any information regarding the source of the violation of the property. Nonetheless, during the unfolding, they will be duplicated, and some of them will be colored in red, and the others in green. As the reader can see on this figure, we performed this alternative coloration, where task *Identify Account Type* and the following ones are colored in yellow. By doing so, we avoid the task duplication induced by the merge gateway, and we are able to to color the initial BPMN process directly and return it to the user.

— 4 —
Implementation

In this chapter, we present the implementation of our approach and two experimental studies carried out to evaluate and validate our approach. It is important to note that everything that has been presented in Chapter 3 has been prototyped and tested, except the ideas presented in section 3.3.

4.1 Tool

The prototype of the approach has been written in Java and consists in almost 10,000 lines of code. It is worth noting that the prototype contains a lot of processing phases which have not been described previously in this report, and which mostly consists in rewriting elements in other formats, such as BPMN to graph, BPMN to AUTX (extended AUT, containing information regarding the faulty states and the color of the transitions, used to encode the CLTS), AUTX to graph, etc.
The prototype allows the user to perform the matching analysis and the unfolding, and if needed the folding of the BPMN process. We take as input the working directory in which the computations will be performed. One can also specify the BPMN source file, the BPMN target file, the AUTX file, the AUT file, and if we should overwrite existing files or not. Finally, we also give to the user the choice of performing or not gateway folding.
Figure 4.1 gives a detailed representation of the lifecycle of the prototype, in which all processing steps have been added.

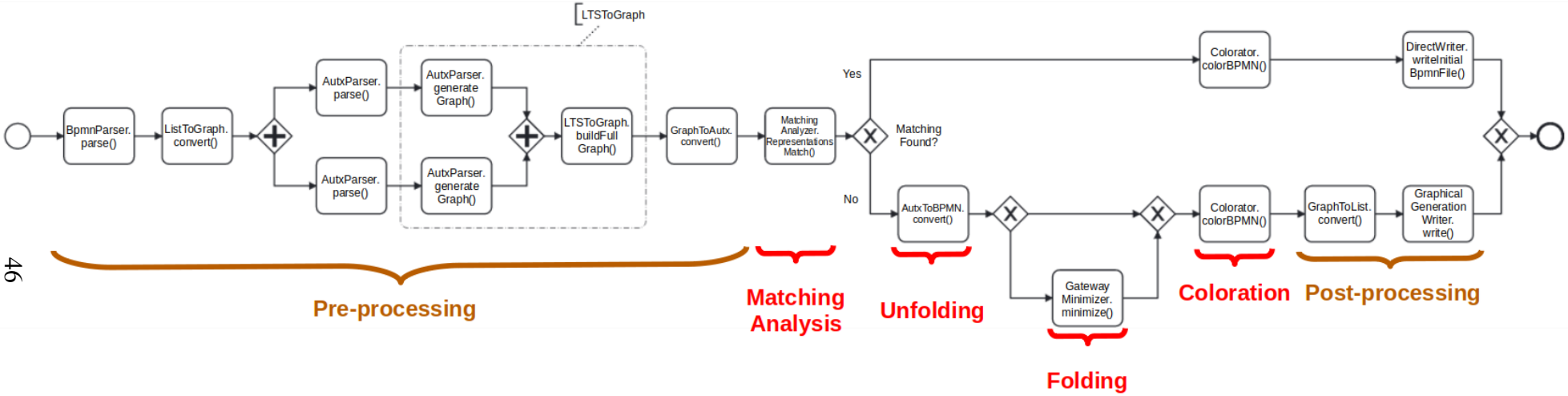

Figure 4.1: Tool implementation workflow

As the reader can see, a lot of processing is needed to handle the different file structures used during the lifetime of the process. One may have noticed the four steps presented in Chapter 3, namely the matching analysis, the unfolding, the folding, and the coloration. The rest of this figure, separated in both *pre-processing* and *post-processing* phases, has not been detailed in this report, as it mostly consists in implementation and formating details.
Nonetheless, it is worth detailing one of these steps, which is the *LTSToGraph.buildFullGraph()* step of pre-processing. In fact, the CLTS built and returned by CLEAR does not contain the entire part of the process that satisfies the property, but is truncated at the first green state. This is an implementation choice, aiming at reducing the number of states of the CLTS, and consequently its size when represented visually. However, in our case, we need the complete this CLTS to perform the matching analysis. Otherwise, it may happen that some red or neutral transitions also belong to the truncated green part of the CLTS. In such cases, the matching would say that no incorrect transition was found, distorting the result. To overcome this, we build the full CLTS with the help of the CLTS output by CLEAR and the initial LTS. Finally, this full CLTS is given as input to the program.

4.2 Experiments

This section aims at evaluating and validating the approach and the implementation. First, we detail the results of an empirical study over 14 real-world examples. In this study, we want to see how the approach behaves, or more precisely how useful the approach could be on such examples. To do so, we compute metrics like the number of BPMN processes directly colorable, or the number of BPMN processes folded after their generation.
Second, we study the performances of the prototype on various examples in which we vary parameters, such as the number of BPMN nodes in the initial graph, or the number of BPMN nodes in the generated graph. This aims at validating the scalability of the prototype on real-world BPMN processes, and at learning the size of the BPMN process, in number of nodes, until which the execution time remains reasonable, according to different criteria.

4.2.1 Empirical Study

For this study, we focus on the direct coloration and the foldability of the BPMN processes considered. These processes were taken from the litterature, and regroup various domains, going from computer sciences through finance and ending with metallurgy. Note that, as we could not find BPMN processes with their corresponding MCL properties, the MCL properties used in this study have been manually written. Nonetheless, we paid a special attention to these MCL properties, by trying to make them as realistic as possible in the context of their BPMN process. In this study, we want to compute the percentage of these processes that are directly colorable, and if they are not, the number of them that have been folded after unfolding. We also decided to put the theoretical results of this study regarding the 2 ideas proposed in Section 3.3.
The results of this study are presented in Table 4.1. It is organized as follows:

- **Column 1** contains the name of the BPMN process.
- **Column 2** indicates if the BPMN process is directly colorable regarding the MCL property.

- **Column 3** indicates if the BPMN process has been folded after unfolding.
- **Column 4** indicates if the BPMN process has rollable loops (Idea 1).
- **Column 5** indicates if the BPMN process is colorable with the color variations method (Idea 2).

Values present in Table 4.1 respect the following notation: Value *X* means *True* while the absence of value means *False*. For the foldability, value *X* may be followed by a value *(a/b)* where *a* represents the number of gateways effectively folded and *b* the number of foldable gateways in the BPMN diagram, manually counted. The mention *N/A* is used in columns 3, 4 and 5 if the current BPMN diagram has been colored directly, meaning that there is nothing to fold, roll, or color differently.

BPMN Example	Directly Colorable	Foldable	Rollable	Colorable with variations
1. Account Opening	X	N/A	N/A	N/A
2. Publication Process	X	N/A	N/A	N/A
3. Credit Offer	X	N/A	N/A	N/A
4. Vacations Booking		X (12/20)		
5. Account Opening		X (8/8)		
6. Plane Entry		X (3/3)		
7. Mortgage Application		X (2/2)		
8. Denoising Process			X	
9. Buying Process			X	
10. Login Process			X	
11. Support Ticket			X	
12. Steel Transformation				X
13. Business Process				
14. Job Hiring				

Table 4.1: Results of the empirical study on real world examples

We know that this study has been made on only few examples. The following results are then to be taken with a grain of salt, as they may not represent exactly the reality. From this table, we can see that 22% of the examples are directly colorable while 78% are not. Among these 78%, 36% are foldable, 36% are rollable, 18% are unrolled and returned as is, and 10% are colorable with variations. Note that, even if it is not the case in this study, examples can be foldable, rollable, and colorable with variations at the same time. By looking at the foldable ones, we can see that 84% of the foldable gateways have been folded after unfolding. Overall, we see that no improvement of the unrolled BPMN process can be made in only 7.8% of cases, and that only 25% of the foldable gateways have not been detected by the tool. Moreover, by verifying manually the non-folded gateways, we see that those gateways are part of the out-of-scope paths of folded gateways. For now, out-of-scope paths are added as is, without any modification. Managing foldable gateways in them is part of future work.
Anyway, these are encouraging results, showing that we are able to perform a reduction of the number of nodes in the generated BPMN process in more than 90% of cases.

From these results, we categorized different patterns:

- **(i)** the processes directly colorable have a property that concerns an exclusive split gateway that is not merged.
- **(ii)** the processes foldable contain parallel gateways with paths of size 1 that have been unfolded during the model checking phase.
- **(iii)** the processes rollable contain loops that have been unrolled during the model checking phase.
- **(iv)** the processes colorable with variations have a property decidable before a merge gateway.
- **(v)** the processes left untouched are processes that contain loop and for which the property concerns one element of the loop that can not reach an end event without traversing a node already traversed.

It is interesting to note that those behaviours are very similar to the ones presented in Chapter 3. Moreover, these behaviours are mostly structural, meaning that one could be able to perform an analysis of the initial BPMN process without using the prototype, and possibly predict the behaviour of the prototype.

4.2.2 Performance Study

In this section, we study the performance of the prototype over a series of examples. Those examples can be decomposed in 3 parts:

- **(i)** real-world examples of the empirical study presented in the previous section.
- **(ii)** examples generated by the tool that contain a high number of nodes in sequence.
- **(iii)** examples generated manually with a few nodes, but high parallelization.

Part 1 aims at verifying the efficiency of the prototype on real-world examples, while parts 2 & 3 aim at verifying the performance of all the toolchain (VBPMN, CLEAR, and the prototype) on large generated examples, in order to assess the scalability of each component.

Before going into the details of this study, we need to describe an implementation part of the gateway folding process. To be able to detect unfolded gateways in the BPMN process, we perform a step called *flattening*, in which we remove all the interleavings between nodes of the BPMN process. To do so, we copy the BPMN process, and each time we cross an already visited node, we duplicate it instead of reusing it. This step is mandatory if we want to be able to detect the size-2 diamond-shaped paths, otherwise we may detect paths that interleaves together and are no longer size-2 diamond-shaped paths. Note that this flattening of the BPMN process induces a huge increase of its number of nodes. This phenomenon is even more remarkable in case of highly parallel BPMN processes, which generates highly interleaved CLTS, for which each state is duplicated several times when converted to BPMN (see Appendix A.1).
Part 3 presents the results of this study over such BPMN processes, which can be considered as limit cases of our approach.

The results of this performance study are presented in Table 4.2.2, which is organized as follows:

- **Column 1** contains the name of the BPMN process.

- **Column 2** contains the number of states in the LTS given as input to CLEAR.
- **Column 3** contains the number of BPMN nodes in our tool, when the user decided not to perform gateway folding.
- **Column 4** contains the number of BPMN nodes in our tool, when the user decided to perform gateway folding.
- **Column 5** contains the time taken by VBPMN to execute (in seconds).
- **Column 6** contains the time taken by CLEAR to execute (in seconds).
- **Column 7** contains the time taken by our tool to execute, when the user decided not to perform gateway folding (in seconds).
- **Column 8** contains the full time taken by the process to execute, when the user decided not to perform gateway folding, including the time taken by VBPMN, CLEAR and our tool (in seconds).
- **Column 9** contains the time taken by our tool to execute, when the user decided to perform gateway folding (in seconds).
- **Column 10** contains the full time taken by the process to execute, when the user decided to perform gateway folding, including the time taken by VBPMN, CLEAR and our tool (in seconds).

Note that, for the examples of the empirical study, only the global time with folding has been computed, as it represents the execution time reached in the classical execution of the tool, where matching and, if needed, unfolding and folding, are performed.
For a better analysis of the results, we defined time intervals for evaluating the execution times. These definitions are based on the fact that this approach is an offline approach, meaning that it can be launched by a user while doing something else, and that results are not needed in real-time. Thereby, we decided that an execution time between 0s and 120s is reasonable (written in classic text), an execution time between 120s and 6000s is long (written in underlined text), and an execution time greather than 6000s is very long (written in underlined bold text).

BPMN Example	Number of BPMN elements	Number of elements in CLEAR	Number of elements without folding	Number of elements with folding	Time taken by VBPMN (s)	Time taken by CLEAR (s)	Time taken by prototype without folding (s)	Global time without folding (s)	Time taken by prototype with folding (s)	Global time with folding (s)
1. Vacations Booking	13								3.436	3.436
2. Account Opening	22								1.209	1.209
3. Publication Process	20								1.351	1.351
4. Steel Transformation	42								1.475	1.478
5. Plane Entry	27								2.235	2.235
6. Mortgage Application	15								1.974	1.974
7. Denoising	16								1.756	1.756
8. Credit Offer	15								1.004	1.004
9. Buying Process	29								1.540	1.540
10. Business Process	15								1.325	1.325
11. Job Hiring	29								1.798	1.798
12. Login Process	17								1.204	1.204
13. Support Ticket	22								1.431	1.431
14. Generated Large 1	62	140	51	62	15.98	0.164	1.254	17.40	1.881	18.03
15. Generated Large 2	126	284	100	126	32.04	2.080	1.262	33.38	1.376	33.50
16. Generated Large 3	254	572	197	254	165.4	6.461	1.790	173.6	2.440	174.2
17. Generated Large 4	510	1148	390	510	902.3	124.3	3.024	**1030**	3.643	**1030**
18. Generated Parallel 1	10	86	27	37	12.21	0.059	0.999	13.27	1.034	13.30
21. Generated Parallel 2	19	529	203	11582	13.07	0.086	1.807	14.96	19.76	32.92
22. Generated Parallel 3	20	606	233	23165	13.15	0.089	2.099	15.34	82.68	95.92
23. Generated Parallel 4	21	683	264	43614	13.23	0.088	2.678	16.00	466.8	480.1
24. Generated Parallel 5	22	770	298	87229	13.61	0.082	4.088	17.78	3948	**3961**
25. Generated Parallel 6	23	857	333	165307	17.81	0.122	6.271	24.20	10100	**10117**

Table 4.2: Results of the performance study on 3 types of BPMN diagrams

Let us now study the results obtained. For the real-world BPMN processes of the empirical study, we see that the worst global execution time is approximately 3s, which is reasonable. This execution time highlights the good performance of the tool over various real-world BPMN processes, and allows us to validate the efficiency of our prototype on BPMN processes consisting of few dozens of nodes.

Now, let us focus on the generated examples. These examples aim at verifying the scalability of the prototype on larger BPMN processes, in terms of number of nodes.

For the large generated examples, we observe a highest execution time of 1030s, or more than 17 minutes, for a BPMN process containing 510 nodes, which is already a lot. In this case, it is interesting to decompose this global execution time to highlight the time taken by each component of the process. Indeed, we see that the main waste of time comes from VBPMN (902s) and CLEAR (124s). We also see that the time taken by our prototype is actually only 3s, which is reasonable. This decomposition shows that our approach is mainly limited by VBPMN and CLEAR (99.7% of time taken) while verifying BPMN processes consisting of few hundreds of nodes.

Now that we have global results concerning real-world examples, and results identifying the limits of VBPMN and CLEAR, we would like to highlight the limits of our prototype. For this purpose, we present the results obtained on the set of generated BPMN processes with high parallelization. Such processes are advantageous for us because they contain only a few nodes, or a few states in their corresponding LTS, which ensures low execution times for VBPMN and CLEAR. However, the flattening phase generates BPMN processes with a huge number of nodes in this case, which will help us studying the limits of our prototype.

By observing the results, we see that the execution time starts being long on BPMN process 22, with a global execution time of 480s. In this example, 466s are taken by the prototype to complete its execution, while only 14s are taken by VBPMN and CLEAR. This execution time is indeed exceeding the threshold of reasonable execution times. But in this case, we see that, starting from a BPMN process consisting of 21 nodes, the flattening phase generated a BPMN process consisting of 43,614 nodes, which is very large. Without any assumption of any kind, one can assert that real-world BPMN processes will never reach such a high number of nodes.

In such conditions, we can conclude that our prototype is performing efficiently for any kind of real-world BPMN process.

— 5 —
Related Work

In this chapter, we present the existing works that are the most related to the problem that we want to solve, and the approach proposed in this report. First, we describe existing works aiming at identifying syntactic problems of BPMN processes using Petri nets. Second, we present works targetting formalization and verification of the semantics of BPMN processes using process algebras. Third, we detail works proposing ways of performing verification and validation of BPMN processes using model checking. Last, we present the work that is the closest to ours, aiming at debugging BPMN processes while expressing the bug(s) in BPMN notation.

Identification of syntactic problems using Petri nets

Several previous works have focused on providing formal semantics and verification techniques for BPMN processes using a rewriting of BPMN into Petri nets, such as [25, 14, 15, 30, 12]. In these works, the focus is mostly on the verification of behavioural or syntactic problems of the BPMN, such as *deadlock* or *livelock*. As far as rewriting logic is concerned, in [17], the authors propose a translation of BPMN into rewriting logic with a special focus on data objects and data-based decision gateways. They provide new mechanisms to avoid structural issues in workflows such as flow divergence by introducing the notion of well-formed BPMN process. Their approach aims at avoiding incorrect syntactic patterns whereas we propose automated analysis at the semantic level. Rewriting logic is also used in [16] for analyzing BPMN processes with time using simulation, reachability analysis, and model checking to evaluate timing properties such as degree of parallelism and minimum/maximum processing times. In this approach, we focus on verification of semantically incorrect BPMN processes.

Formalization and verification of BPMN processes using process algebra

Let us now concentrate on those using process algebras for formalizing and verifying BPMN processes, which are closer to the approach proposed in this report. The authors of [34] present a formal semantics for BPMN by encoding it into the CSP process algebra. They show in [35] how this semantic model can be used to verify compatibility between business participants in a collaboration. This work was extended in [33] to propose a timed semantics of BPMN with delays. [10, 27] focus on the semantics proposed in [34, 33] and propose an automated transformation from BPMN to timed CSP. In [19] the authors have proposed a first transformation from BPMN to LNT, targeted at checking the realizability of a BPMN choreography. In [12],

the authors propose a new operational semantics of a subset of BPMN focusing on collaboration diagrams and message exchange. The BPMN subset is quite restricted (no support of the inclusive merge gateway for instance) and no tool support is provided yet. Compared to the above approaches, ours focuses on verification of BPMN orchestrations while theirs focus on BPMN choreographies. We also allow a larger BPMN subset and propose a tool support along with automated techniques.

Verification and validation of BPMN processes using model checking

The works proposed in [23, 21, 28] are close to the approach proposed in this work. Indeed, the approach proposed in [23] proposes verification and comparison techniques based on model checking. The verification part is decomposed in 3 steps. Step 1 consists in translating the initial BPMN process into a semantically equivalent model called Process Intermediate Format (PIF). From this PIF model, step 2 has the purpose of translating it into LNT, and then map it to LTS. Step 3 gives the LTS and the MCL property to a model checker for verification, and returns a counterexample violating the property, if it exists. This approach is close to ours, in the sense that it allows the user to verify a safety temporal logic property over a BPMN process, while providing him a counterexample in the form of an LTS if the property is violated. Other works, such as [21] and [28] are also making use of model checking to perform verifications of temporal logic properties over BPMN processes. The main differences with our approach are the format of the counterexample returned (LTS against BPMN), and the number of counterexamples returned. Indeed, the approaches detailed here propose a single counterexample, while ours returns a representation of all the counterexamples of the property.

Debugging of BPMN processes

In this paragraph we discuss the approach proposed in [31]. This approach aims at representing visually violations of temporal logic properties regarding a given BPMN process. The authors oriented the work on what they called *containment checking*. Containment checking consists in verifying properties built from high-level BPMN processes (*e.g.* representations of BPMN processes without much details) over low-level models (*e.g.* complete representation of BPMN processes). In this work, the high-level BPMN process is converted into linear temporal logic (LTL) properties [24] while the low-level model is translated into SMV [8]. Both are given as input to the nuSMV [11] model checker which generates counterexamples if the property is violated. Then, the results are given to the visualization engine that represents, in BPMN notation, the counterexamples given by nuSMV. The representation of the bad behaviours is close to the one proposed in our approach, as elements satisfying the property are colored in green and those violating the property in red.
There are three main differences with our approach. The first one is the class of property managed. In this approach, we want to verify the conformance of a low-level BPMN process with regards to a high-level BPMN model. This is comparable to the verification of conformance of a model compared to its meta-model in software engineering. In our case, we verify safety properties of all kinds. The second one is that in their case, they propose countermeasures to the bugs found. Indeed, given a set of elements and an LTL property, they propose 2 or 3 countermeasures to avoid the bug. Along with the coloration of the BPMN process, they add annotations to the BPMN process at the location of the bug, which detail the possible counteremeasures. Nonetheless, no countermeasure is proposed for cases that do not match their

conditions. The last one is that they do not present the cases where the counterexamples are not representable on the initial BPMN process. In our case, we generate a new BPMN process and we propose different solutions to make it as close syntactically as the initial one.

— 6 —
Conclusion

In this research project, we proposed a way of improving and simplifying the comprehension and the visualization of behavioural problems in BPMN processes regarding some safety temporal logic properties. The main objectives of this approach were **(i)** to give a visual feedback of the problem expressed in BPMN notation and **(ii)** to stay as syntactically close as possible to the original BPMN process while remaining semantically equivalent, and **(iii)** to be as minimal as possible in terms of BPMN nodes in the final process. Guided by this purpose of clarity, we chose the coloration of BPMN processes as visualization technique.
To do so, we first presented a solution in which the original BPMN process of the user is preserved, and colored according to the counterexamples exhibited by the model checker. However, even if this is the best solution, because no modification of the original BPMN process is performed, there are cases in which it is not appliable. These cases have been identified and detailed in this report. To manage them, we proposed a second approach which, in a first step, generates a new BPMN process from the CLTS output by CLEAR (*unfolding*), and in a second step, performs some minimization over it to lower its number of nodes (*folding*). Nonetheless, it may happen that the initial BPMN process is not directly colorable, and the generated BPMN process can not be minimized. In this case, we give to the user the generated BPMN process, colored. This solution is in general satisfying, but can in some cases make the generated BPMN process large. However, such cases do not happen often, and proposing solutions to improve them is part of future work.
To finish, we made a prototype in Java, consisting of almost 10,000 lines of code. In order to verify the usability and the scalability of this prototype, we performed two studies: an empirical one to get insights of the behaviour of this approach on real-world examples, and a performance one aiming at ensuring that the prototype runs in reasonable time on real-world examples, and at verifying its scalability on large BPMN processes.

Future Work

According to the current state of this project, some improvements are still to be done. The three main ones are outlined below:

1. In a first step, we would like to handle inclusive gateways in the foldability process. For now, only parallel gateways are supported by this process. Nonetheless, even if they are less common in BPMN processes, it is interesting to handle them. Moreover, inclusive

and parallel gateways are only slightly different, and we strongly believe that the existing algorithm would need only few refinements to handle those gateways.

2. In a second step, we would like to focus on liveness properties. Indeed, this work only takes as input safety properties, but not liveness properties, which represent an important part of temporal logic properties. Furthermore, liveness properties and safety properties could be processed similarly in this approach. Indeed, the CLEAR tool works with liveness properties too, so one can imagine the same kind of visual representation than the one presented in this approach, where red parts would represent the violation of the property, and green parts its satisfaction.

3. Finally, we would like to improve the folding of the parallel gateways, particularly to handle gateways with inner paths of size 2 or more, or containing loops. Ideas have already been proposed, especially concerning the folding algorithm currently used. We believe that modifications of this algorithm could lead to significant improvements regarding the number of gateways detected and folded. Consequently, this would induce a decrease of the size of the BPMN processed returned.

— A —
Appendix

A.1 Flattening of a highly parallel BPMN process

Figure A.2 shows an example of meshed LTS, built from the highly parallel BPMN process visible in Figure A.1. Such a CLTS is then flattened during the generation of the new BPMN process, which results in the BPMN process proposed in Figure A.3.

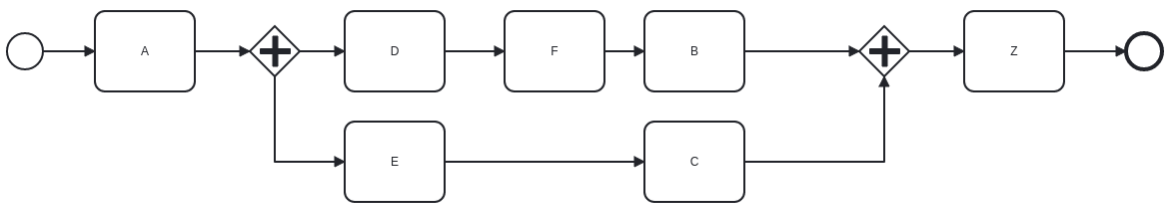

Figure A.1: Highly parallel BPMN process

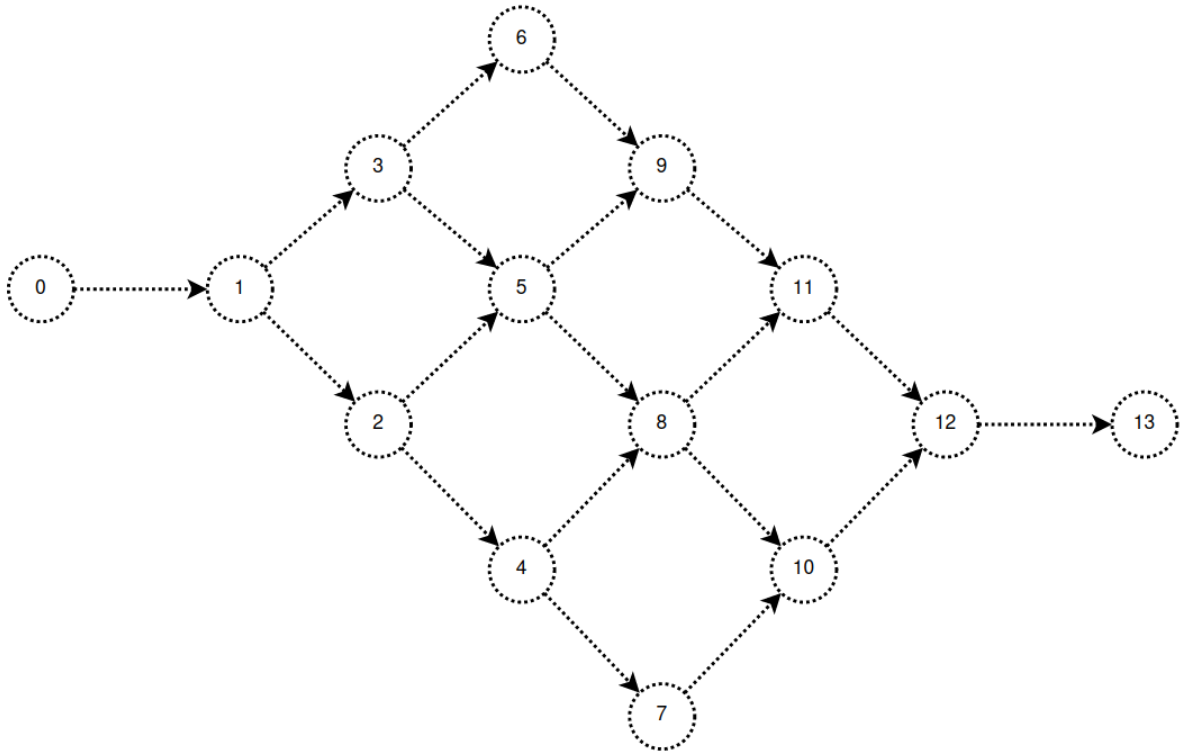

Figure A.2: Meshed CLTS containing 14 states

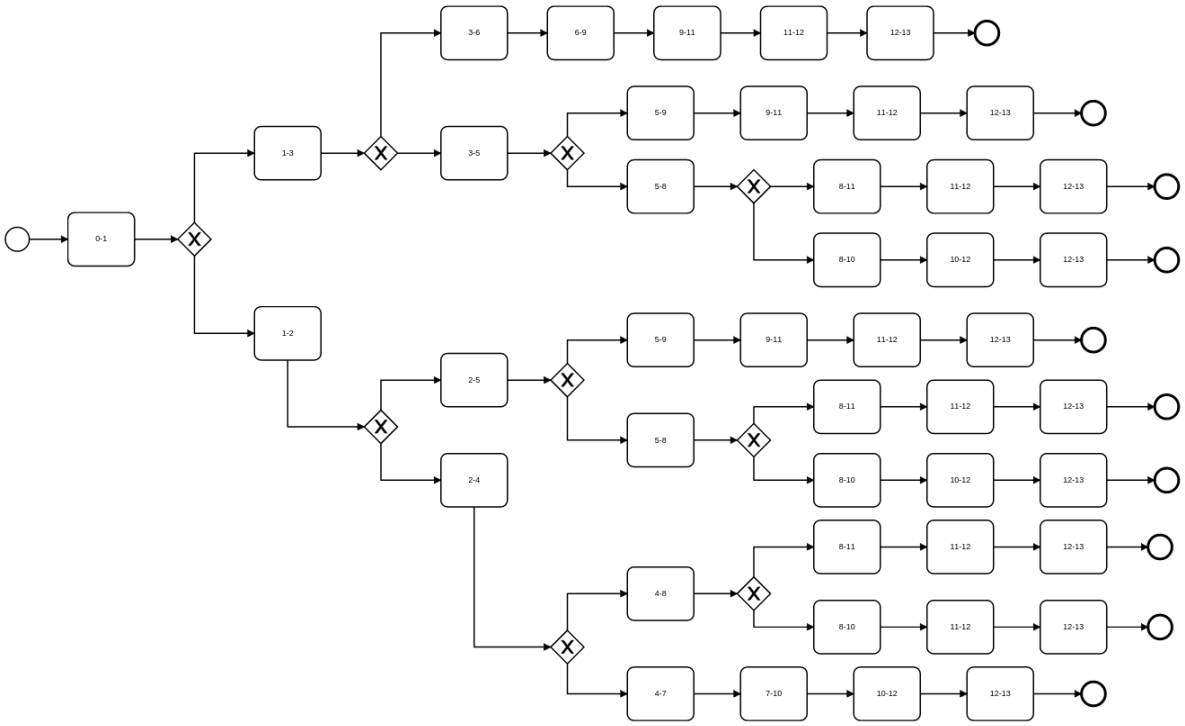

Figure A.3: 64 nodes BPMN process generated after flattening

A.2 Transformation of a CLTS into a semantically equivalent BPMN process

Given any CLTS (Figure A.4), we are able to generate the semantically equivalent BPMN process (Figure A.5). Note that some BPMN elements have no representation in the CLTS, such as the events. To generate a valid BPMN process, we add an initial event before its initial node, and end events after all nodes without child.

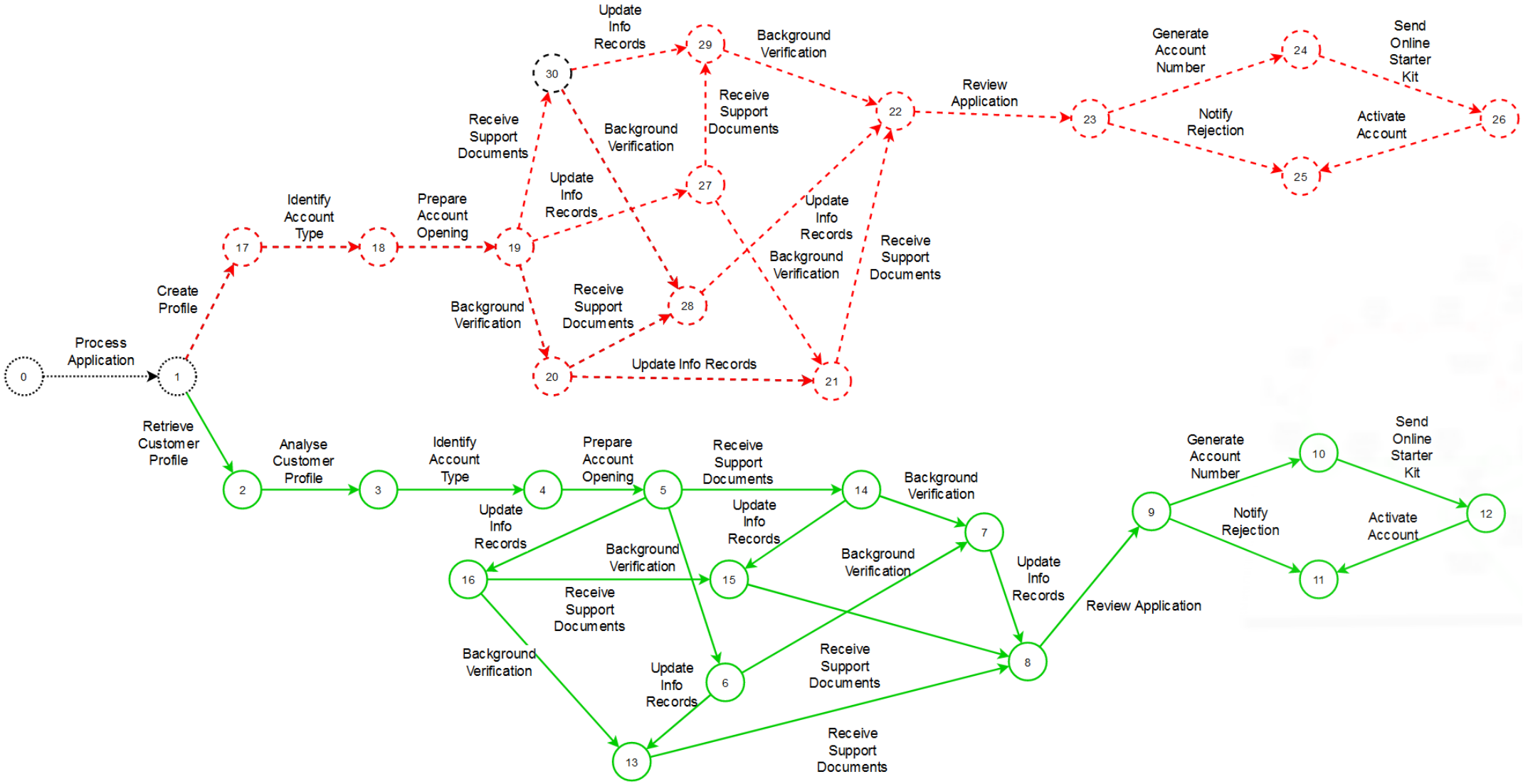

Figure A.4: Example of CLTS

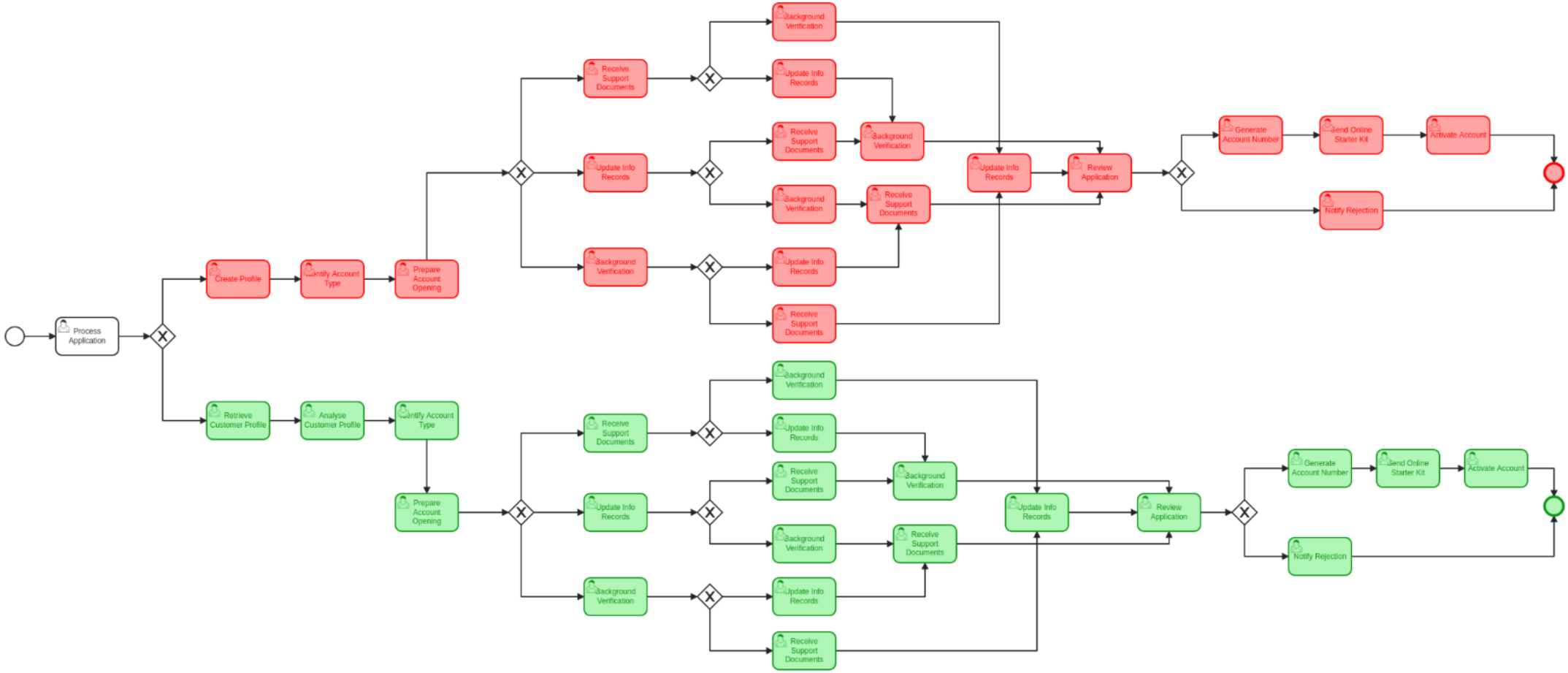

Figure A.5: BPMN generated from the CLTS in Figure A.4

Bibliography

[1] CLEAR Visualization tool. https://gbarbon.github.io/clear/.

[2] VBPMN Verification tool. https://pascalpoizat.github.io/vbpmn-web/documentation.html.

[3] Information technology - Object Management Group Business Process Model and Notation. 2013.

[4] Christel Baier and Joost-Pieter Katoen. *Communication and Concurrency*. MIT Press, 2008.

[5] Gianluca Barbon, Vincent Leroy, and Gwen Salaün. Debugging of Concurrent Systems Using Counterexample Analysis. In *Proc. of FSEN'17*, volume 10522 of *LNCS*, pages 20–34. Springer, 2017.

[6] Gianluca Barbon, Vincent Leroy, and Gwen Salaün. Debugging of Behavioural Models with CLEAR. In *Proc. of TACAS'19*, volume 11427 of *LNCS*, pages 386–392. Springer, 2019.

[7] Gianluca Barbon, Vincent Leroy, Gwen Salaün, and Emmanuel Yah. Visual Debugging of Behavioural Models. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proc. of ICSE'19*, pages 107–110. IEEE / ACM, 2019.

[8] Béatrice "Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, Philippe Schnoebelen, and Pierre" Mckenzie. *"SMV — Symbolic Model Checking"*, pages "131–138". "Springer Berlin Heidelberg", "Berlin, Heidelberg", "2001".

[9] Francesco Calzolai, Rocco De Nicola, Michele Loreti, and Francesco Tiezzi. TAPAs: A Tool for the Analysis of Process Algebras. *Trans. Petri Nets and Other Models of Concurrency I*, 5100:54–70, 2008.

[10] M. I. Capel and L. E. Mendoza Morales. Automating the Transformation from BPMN Models to CSP+T Specifications. In *Proc. of SEW*, pages 100–109. IEEE, 2012.

[11] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, pages 410–425, 2000.

[12] F. Corradini, A. Polini, B. Re, and F. Tiezzi. An Operational Semantics of BPMN Collaboration. In *Proc. of FACS'15*, volume 9539 of *LNCS*, pages 161–180. Springer, 2015.

[13] Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. An Overview of the mCRL2 Toolset and Its Recent Advances. In *Proc. of TACAS'13*, volume 7795 of *LNCS*, pages 199–213. Springer, 2013.

[14] G. Decker and M. Weske. Interaction-centric Modeling of Process Choreographies. *Information Systems*, 36(2):292–312, 2011.

[15] R.M. Dijkman, M. Dumas, and C. Ouyang. Semantics and Analysis of Business Process Models in BPMN. *Inf. Softw. Technol.*, 50(12):1281–1294, 2008.

[16] Francisco Durán and Gwen Salaün. Verifying Timed BPMN Processes using Maude. In *Proc. of COORDINATION*, volume 10319 of *LNCS*, pages 219–236. Springer, 2017.

[17] N. El-Saber and A. Boronat. BPMN Formalization and Verification using Maude. In *Proc. of BM-FA*, pages 1–8. ACM, 2014.

[18] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. In *International Journal on Software Tools for Technology Transfer*, pages 89–107. Springer, 2013.

[19] M. Güdemann, P. Poizat, G. Salaün, and L. Ye. VerChor: A Framework for the Design and Verification of Choreographies. *IEEE Trans. Services Computing*, 9(4):647–660, 2016.

[20] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. LTSmin: High-Performance Language-Independent Model Checking. In *Proc. of TACAS'15*, volume 9035 of *LNCS*, pages 692–707. Springer, 2015.

[21] Oussama Kherbouche, Adeel Ahmad, and Henri Basson. Using model checking to control the structural errors in BPMN models. *IEEE International Conference on Research Challenges in Information Science (RCIS)*, 7, 2013.

[22] Ajay Krishna, Pascal Poizat, and Salaün Gwen. Checking Business Process Evolution. In *Science of Computer Programming*, pages 1–26. Elsevier, 2019.

[23] Ajay Krishna, Pascal Poizat, and Gwen Salaün. VBPMN: Automated Verification of BPMN Processes. *13th International Conference on integrated Formal Method*, 2017.

[24] Fred Kröger and Stephan Merz. *Temporal Logic and State Systems*. Springer Berlin, Heidelberg.

[25] A. Martens. Analyzing Web Service Based Business Processes. In *Proc. of FASE'05*, volume 3442 of *LNCS*, pages 19–33. Springer, 2005.

[26] Radu Mateescu and Damien Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of FM'08*, volume 5014 of *LNCS*, pages 148–164. Springer, 2008.

[27] L. Mendoza-Morales, M. Capel, and M. Pérez. Conceptual Framework for Business Processes Compositional Verification. *Inf. & Sw. Techn.*, 54(2):149–161, 2012.

[28] Toufik Messaoud Maarouk, Mohammed El Habib Souidi, and Nadia Hoggas. Formalization and Model Checking of BPMN Collaboration Diagrams with DD-LOTOS. *Computing and Informatics*, 40, 2021.

[29] Robin Milner. *Principles of Model Checking*. Prentice Hall International, 1989.

[30] I. Raedts, M. Petkovic, Y. S. Usenko, J. M. van der Werf, J. F. Groote, and L. Somers. Transformation of BPMN Models for Behaviour Analysis. In *Proc. of MSVVEIS'07*, pages 126–137, 2007.

[31] Faiz UL Muram, Huy Tran, and Zdun Uwe. Counterexample Analysis for Supporting Containment Checking of Business Process Model. *International Workshop on Process Engineering (IWPE)*, 1, 2015.

[32] Wil van der Aalst. *Process Mining: Data Science in Action*. Springer, 2016.

[33] P. Y. H. Wong and J. Gibbons. A Relative Timed Semantics for BPMN. *Electr. Notes Theor. Comput. Sci.*, 229(2):59–75, 2009.

[34] P.Y.H. Wong and J. Gibbons. A Process Semantics for BPMN. In *Proc. of ICFEM'08*, pages 355–374, 2008.

[35] P.Y.H. Wong and J. Gibbons. Verifying Business Process Compatibility. In *Proc. of QSIC'08*, pages 126–131, 2008.