Debugging of BPMN Processes Using Coloring Techniques

Quentin Nivon, Gwen Salaün

Debugging of BPMN Processes using Coloring Techniques

Quentin Nivon and Gwen Salaün

Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, F-38000 Grenoble France

Abstract. A business process is a collection of related tasks organized in a specific order whose overall execution solves a specific service or product. BPMN has become the standard workflow-based notation for developing business processes. Designing business processes using BPMN is however error-prone. Recent works have proposed verification techniques for analyzing processes and for detecting possible issues. In particular, model checking is an established technique for automatically verifying that a model (e.g., a BPMN process) satisfies a given temporal property. When the model violates the property, the model checker returns a counterexample, which is a sequence of actions leading to a state where the property is not satisfied. Understanding this counterexample for debugging the process is not an easy task, especially if the counterexample is not expressed using the original notation (BPMN here). In this paper, we focus on the model checking of BPMN processes. When properties are violated, we propose to transform counterexamples back on to the original BPMN process in order to simplify the debugging steps. To do so, we rely on coloration techniques. The approach proposed in this paper is fully automated using several tools and was validated on many examples.

1 Introduction

For decades, 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 graphically describe 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 counterexample can be complex, even for expert users. It is even more complex for beginners and BPMN 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.

In this paper, we focus on the model checking of BPMN processes. The main goal of this work is to transform counterexamples back on to the original BPMN process in order to simplify the debugging steps. To do so, we rely on some already existing techniques. The first one, called VBPMN [19], is used for converting BPMN into LNT. The second one is model checking [25], that we use for analyzing and debugging LNT specifications with respect to some temporal logic properties. The last one, called CLEAR [3] is used for generating a colored LTS representing the full set of traces violating the property, called a counterexample LTS (CLTS) [3]. This CLTS can however be hard to understand, for several reasons: (i) the CLTS notation strongly differs from the BPMN notation, (ii) the CLTS can be quite large 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 our approach is to transform the information available in the CLTS model on to the BPMN model for simplifying the debugging phase. To do so, the approach consists of three cases or steps, which are handled one after the other. The first step, 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 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 second step is called *unfolding*. 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 by restructuring parts of the process using, for instance, parallel gateways. This third step preserves the semantics of the input process and is called *folding*.

Note 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. Safety properties state that *"something bad must not happen"*.

Figure 1 gives an overview of the contributions presented in this paper. The approach takes as input the initial BPMN process and the temporal logic property to verify. The first step is performed by the VBPMN tool [19], which takes the BPMN process given as input and transforms it into LNT. Then, the LNT specification and the temporal logic property are given to CLEAR [5], 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, the BPMN process is colored according to the CLTS. Otherwise, we generate a new BPMN process with the help of the CLTS, we try to fold it, and finally we color the resulting process.

All the steps of the approach are fully automated by a prototype tool we implemented. As for evaluation, we carried out two experimental studies in order to assess the usability and performance of the approach on real-world and large examples.

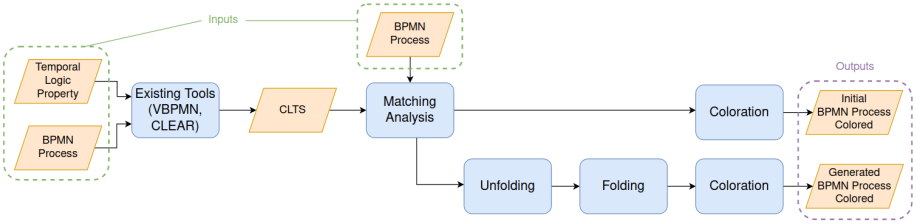

Fig. 1: Overview of the approach

The rest of this paper is organized as follows. Section 2 introduces several useful notions regarding BPMN and model checking. Section 3 presents the main steps of our approach for visualizing counterexamples at the BPMN level. Section 4 describes the tool support and experiments. Section 5 surveys related work and Section 6 concludes the paper.

2 Background

In this section, we introduce some notions needed to understand this work. First, we describe BPMN, which serves as input model to our approach. Then, we give an overview of Labelled Transition Systems and model checking. Finally, we explain what is a Counterexample LTS (CLTS).

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, and to make it understandable to every one (business analysts, developers, end users). This notation has been first introduced in 2004 by the Business Process Management Initiative and has become an ISO/IEC standard in 2013 [1]. It is now widely used across the world. The current version of BPMN is BPMN 2.0 and more details about it can be found in [1].
In this work, we focus on a subset of the BPMN syntax. This subset, described in Figure 2, takes into account initial and end events, tasks and gateways. A gateway is either a split or a merge. A split gateway consists of a single incoming flow and multiple outgoing flows. A merge gateway consists of multiple incoming flows and a single outgoing flow. Several types of gateways are available, such as exclusive, parallel, and inclusive gateways. An exclusive gateway corresponds to a choice among several flows. A parallel gateway executes all possible flows at the same time. An inclusive gateway executes one or several flows. A study made in [18] on more that 800 real-world BPMN processes shows that the subset used in this paper suffices to build more than 90% of these real-world BPMN processes.

Definition 1. *(BPMN Graph) A BPMN process can be represented as a graph* $B=(S_N,S_E)$ *where* S_N *is a set of nodes (tasks, event, gateways, ...) and* S_E *is a set of edges (sequence flows).*

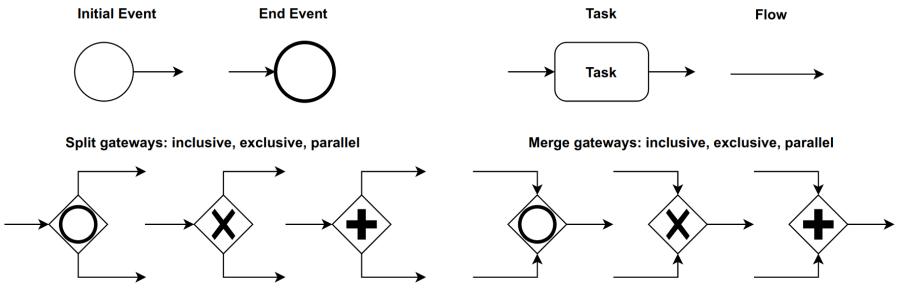

Fig. 2: Supported BPMN syntax

In this work, a Labelled Transition System (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. An LTS is represented by a set of states, and a set of transitions connecting these states.

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 can be seen as the semantical model of a BPMN process. Thus, both models have the same semantics. Indeed, all the elements of the BPMN process have a corresponding element in the LTS, and vice-versa. The correspondence is the following: each state of the LTS is either a sequence flow or an exclusive gateway in the BPMN process, and each transition is a task in the BPMN process. 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* $l_1, l_2, ..., l_n \in \Sigma$ *such that* $s^0 \xrightarrow{l_1} s_1 \in T, s_1 \xrightarrow{l_2} s_2 \in T, ..., s_{n-1} \xrightarrow{l_n} s_n \in T$*. A trace is either infinite because of loops or finite when the last state* s_n *has no outgoing transitions. The set of all traces of* M *is written as* $t(M)$*.*

Model checking consists in verifying that a behavioural model or specification (LOTOS New Technology (LNT) [9] in this work) satisfies a given temporal property P, which specifies some expected requirement of the system. Temporal properties are usually divided into two main families: safety and liveness properties [25]. In this work, we focus on safety properties, which are widely used in the verification of real-world systems. Safety properties state that *"something bad must not happen"*. A safety property is usually formalized using a temporal logic. We use MCL (Model Checking Language) [22] in this work, which is an action-based, branching-time temporal logic suitable for expressing properties of concurrent systems. MCL is an extension of alternation-free μ-calculus with regular expressions, data-based constructs, and fairness operators. A safety property can be semantically characterized by an infinite set of traces t_P, corresponding to the traces that violate the property P in an

LTS. If the LTS model does not satisfy the property, the model checker returns a *counterexample*, which is one of the traces belonging to t_P.

Definition 4. *(Counterexample) Given an LTS* $M=(S,s^0,\Sigma,T)$ *and a property* P*, a counterexample is any trace which belongs to* $t(M)\cap t_P$*.*

The approach presented in [3, 6] 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 only belong to paths in the model that represent behaviours which always satisfy the property.
- *incorrect transitions*, which only 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 a *tagged LTS*.

Definition 5. *(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, which are those 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 6. *(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* $t''=s\xrightarrow{(l,\gamma)}s'''\in T_T$*.*

By looking at outgoing transitions of a faulty state, we can identify four categories of faulty states (Figure 3):

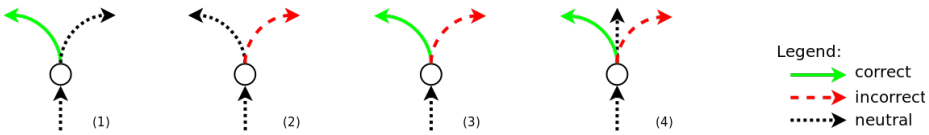

Fig. 3: The four types of faulty states

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.

Finally, a CLTS can be defined as a tagged LTS with faulty states.

3 BPMN Coloration

In this section, we first give an overview of the whole approach for simplifying the debugging of BPMN processes. Second, we present in details the *folding* step, because it is the most complex step of the approach. More details about the whole approach can be found in [26].

3.1 Overview

In this subsection, we give an overview of the three main steps of the approach, which are summarized in Figure 1.

Matching Analysis. The goal of this first step is to determine whether the counterexamples of the temporal logic property appearing in the CLTS can be represented directly on the initial BPMN process. This verification proceeds in two steps. In the first step, we verify whether all the transitions of the CLTS having the same label have the same color. Such a situation can happen whenever we reach a BPMN task that can either violate and satisfy the property, regarding its position in the execution flow. If it is not the case, we know that the initial BPMN process can not be colored directly. Indeed, if two transitions with the same label have different colors, the corresponding (unique) task of the BPMN process should have two different colors at the same time, which is not possible. To solve this issue, we duplicate this task, making by the same time the initial BPMN process not colorable directly. Otherwise, we perform the second step. In this step, we try to associate each faulty state of the CLTS to a unique node of the BPMN process. Such an association is found whenever a BPMN node has ancestor tasks (resp. descendant tasks) for which labels correspond to the labels of the incoming transitions (resp. outgoing transitions) of the current faulty state, and the current BPMN node has not already been associated to a faulty state. If this association is found for each faulty state of the CLTS, then a matching

exists between the BPMN process B and the CLTS M, noted $Match(B,M)$. Then, we color the initial BPMN process, and return it to the user.
Figure 4 illustrates this second step. Note that the four faulty states, namely states 3, 4, 5 and 6, have been mapped to four unique nodes of the initial BPMN process. Thus, a matching was found between the initial BPMN process and the CLTS, so the initial BPMN process is colored and returned to the user. If no such matching can be found, we perform a step called *unfolding*.

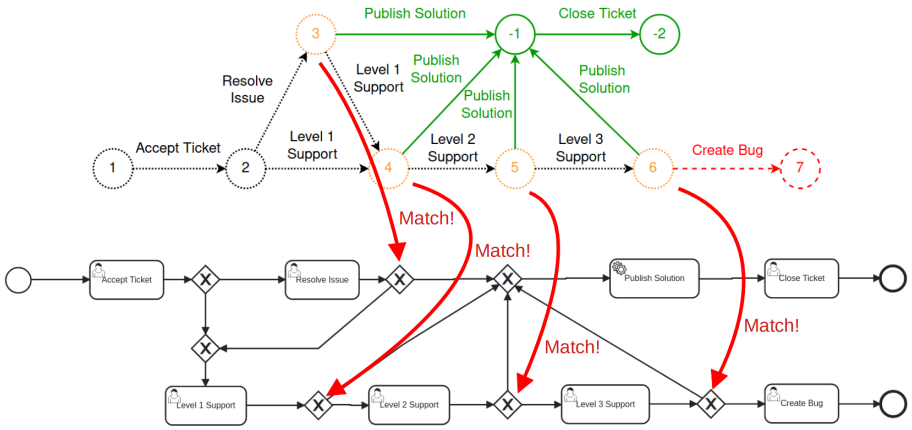

Fig. 4: Example of matching between the CLTS and the BPMN process

Unfolding. In this second step, we generate a new BPMN process that is semantically equivalent to the initial one. This equivalence is preserved because each element of the CLTS has a corresponding element in the BPMN process. More precisely, a transition in the CLTS corresponds to a task in the BPMN process, and a state in the CLTS corresponds either to a sequence flow or to an exclusive split gateway in the BPMN process. As this generated BPMN process is semantically equivalent to the CLTS, and the CLTS is colored, this BPMN process has the advantage of being colorable. Nonetheless, some constructions in the initial BPMN process may generate large BPMN processes during this phase, such as parallel gateways that will be rewritten in their exclusive versions according to Milner's theorem [2] [1]. This phenomenon is accentuated by an implemental step called *flattening*, which occurs during the unfolding. In this step, each state of the CLTS having more than one incoming transition will be duplicated. More precisely, if the CLTS contains states s such that $\exists s' \xrightarrow{l_1} s$ and $s'' \xrightarrow{l_2} s$, then the flattening step will generate a new state s''' equivalent to s such that $s'' \xrightarrow{l_2} s'''$, and discard the transition $s'' \xrightarrow{l_2} s$. The flattening is necessary to be able to detect unfolded parallel gateways and replace them by their folded version, while

[1] A parallel execution of two actions 'a' and 'b' for instance is equivalent to a choice between executing 'a' followed by 'b' or 'b' followed by 'a'.

remaining semantically equivalent to the initial BPMN process. Figure 5 illustrates the unfolding phase, along with the flattening step. As the reader can see, each transition of the CLTS has become a BPMN task, while each state has become a sequence flow or an exclusive gateway. Moreover, one can see that the number of nodes in the generated BPMN process is greater than the number of states and transitions in the CLTS. To mitigate this state explosion, we perform a third and last step called *folding*.

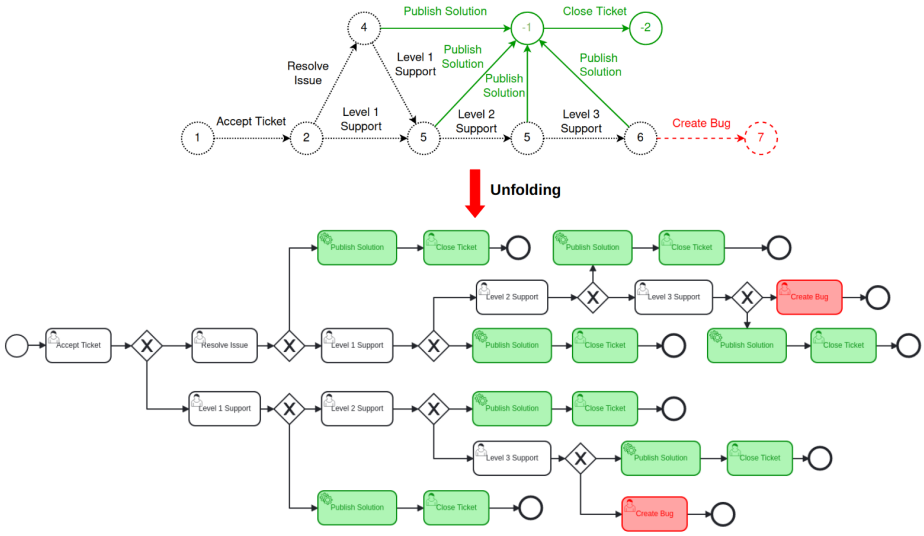

Fig. 5: Example of generation of a BPMN process from a CLTS (Unfolding)

Folding. This third and last step aims at reducing the size (in terms of number of nodes) of the generated BPMN process, while increasing the understandability of this process. To do so, we focus on the detection of specific patterns, that we call *interleavings* or *diamonds*. Figure 6 illustrates with an example of such diamond.

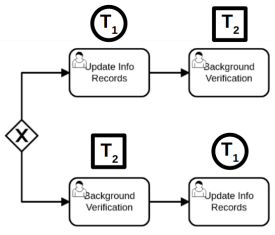

Fig. 6: Example of diamond pattern

As the reader can see, the first task of the upper path corresponds to the second task of the bottom path, and vice versa. In fact, such diamonds result of the rewriting of parallel gateways into their exclusive versions, according to Milner's theorem. Milner's theorem also states that the reverse transformation is possible, allowing us to rewrite this diamond into a parallel gateway containing tasks T_1 and T_2. This transformation may seem of minor interest in this example, as it only reduces the number of tasks from four to two. However, detecting unfolded parallel gateways containing three or more elements may lead to significant reductions of the total number of tasks of the process. Once such gateways have been detected, we generate their semantically equivalent folded version, and we replace the unfolded version by the folded one in the generated BPMN process. This transformation is of prime interest, because it reduces the number of nodes of the BPMN process, while making it syntactically closer to the initial one, and consequently, more understandable for the user.

3.2 Folding

In this section, we focus on the folding step, which is the most crucial and complex one in our approach. In the following, only exclusive gateways are considered, as inclusive and parallel gateways have been rewritten in their exclusive versions according to Milner's theorem. In this context, the word *gateway* will abusively refer to exclusive gateways all along this subsection. Second, the term *folding* will be used to refer to the transformation of a set of tasks belonging to an exclusive gateway into 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 another 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 7. *(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.* $e_n=g$, $\exists i \in [1...(n-1)]$ *where* e_i *is a split gateway and* $\exists p'=(e'_1,...,e'_m)$ *starting from* e_i *for which* $g \in p'$.

Definition 8. *(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* g *is not a nested gateway.*

Figure 7 illustrates nested gateways. As the reader can see, gateways SG2 and MG2, circled in red, are nested gateways, because they are part of the gateways SG1 and MG1. Conversely, these last are outer gateways, are they are not included in any other gateways.
The folding approach proposed is composed of four main steps:

- **Step 1** retrieves 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.

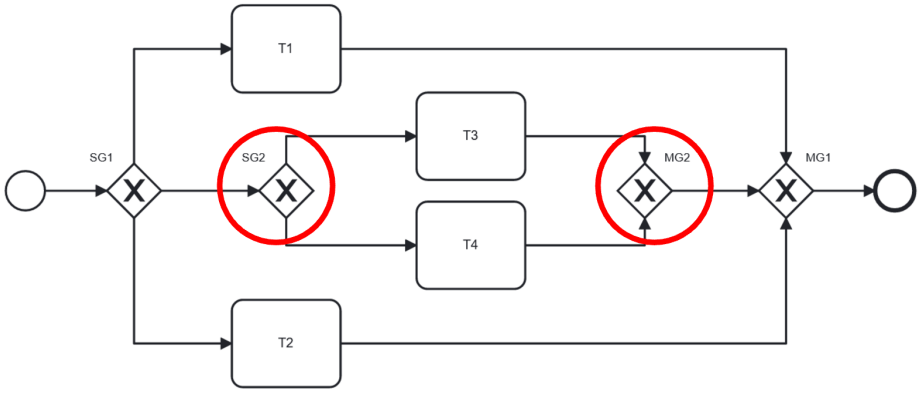

Fig. 7: Example of nested gateways

- **Step 2** computes, for each gateway of each group, a positive integer value representing the foldability that it can not exceed, *i.e.* the maximum size of the equivalent parallel gateway. We call this information *maximum foldability*. This information is then put in a data structure called *metadata*, that will contain information about the foldability of each gateway of the BPMN process.
- **Step 3** fills the metadata of each gateway of each group, in order to know if it is reducible, and, if yes, what is its composition.
- **Step 4** builds 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.

In the rest of this section, we focus on **Step 3**, which is the core step of this approach. Indeed, this step determines if and how each gateway of the generated BPMN process can be folded. Before going further, it is worth noticing that the notion of *maximum foldability* represents the maximum size of the equivalent folded gateway. This notion is a maximum because, in practice, the effective foldability of a gateway can be lower than its maximum foldability, or even null if the gateway is not foldable.
Now, let us detail **Step 3**. **Step 3** is decomposed into two phases: one for gateways of maximum foldability 2, and one for gateways of maximum foldability strictly higher than 2. In this paper, we detail the first phase, managing gateways of maximum foldability 2. In this phase, we analyse all the paths starting from the current gateway in order to extract three pieces of 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 paths. The outgoing paths is a set containing all 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. During this step, we check whether there exists 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*.

Proposition 1. *(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 6 illustrates a gateway having size-2 diamond-shaped paths. Note that 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 three 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 two sets, and remove the first two tasks from each path of each set. The remaining paths in each set are called *out-of-scope paths*.

Proposition 2. *(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)* $\geq$ *3, then the path* $p'=(e_3,...,e_n)$ *is an out-of-scope path.*

Figure 8 shows an example of four out-of-scope paths belonging to two distincts sets of out-of-scope paths. One can see that round tasks and square tasks form two distinct out-of-scope paths, belonging to the same set. The same remark applies to triangle and diamond tasks.

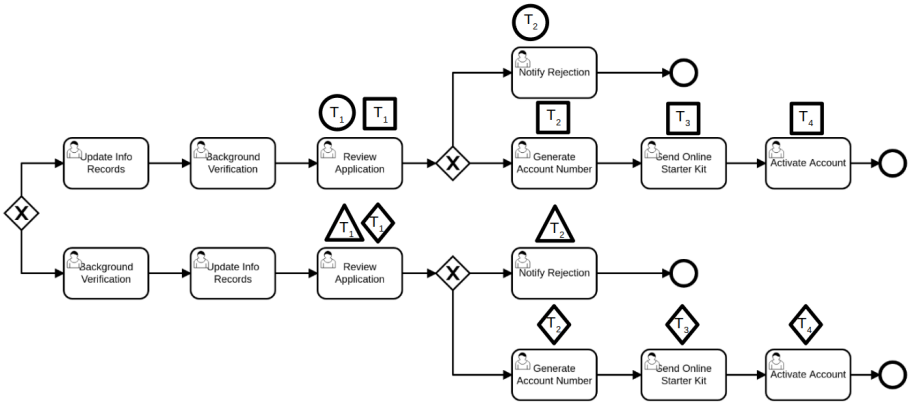

Fig. 8: Example of gateway having 2 sets of 2 out-of-scope paths

Once we have computed those two 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 mark this gateway as foldable in a parallel gateway of size 2. The tasks belonging to this parallel gateway correspond 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 9. *(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 9 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 can be reduced to a parallel gateway of size 2 containing the tasks *Update Info Records* and *Background Verification*, so the path circled in red on the figure is an impure path.

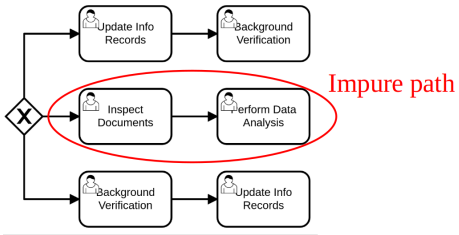

Fig. 9: Example of gateway having 1 impure path

When all this information has been computed, we know precisely if and how each gateway of maximum foldability 2 is foldable. Then, we make use of this information to compute the foldability of gateways of maximum foldability 3, then 4, and so on. For these gateways, the approach is slightly different than the one we have presented above. Indeed, in such cases, we make use of the knowledge we have about the foldability of the inner gateways of maximum foldability lower than the maximum foldability of the current gateway by 1 to know whether it is foldable. A gateway of maximum foldability $f \geq 3$ is foldable if and only if it has f children and for which each child is parent of a pure gateway of maximum foldability $f-1$ containing all the children of the initial gateway except the current child. Figure 10 shows equivalence relationships between gateways illustrating how this step works in practice. As the reader can see, the first picture contains only exclusive gateways. After computing information about gateways of maximum foldability 2, we are able to generate the BPMN process shown in the second picture. With this knowledge, we understand that the whole gateway is a parallel gateway containing three tasks that have been unfolded. Thus, we are able to fold it back, as shown in picture 3.

Now, we have a full representation of the foldabilities that can be processed, thanks to the metadata. Figure 11 illustrates this knowledge by giving a textual representation of the metadata of the 3 gateways G_1, G_2 and G_3 visible on the figure. Following this information, we fold the foldable gateways and replace them

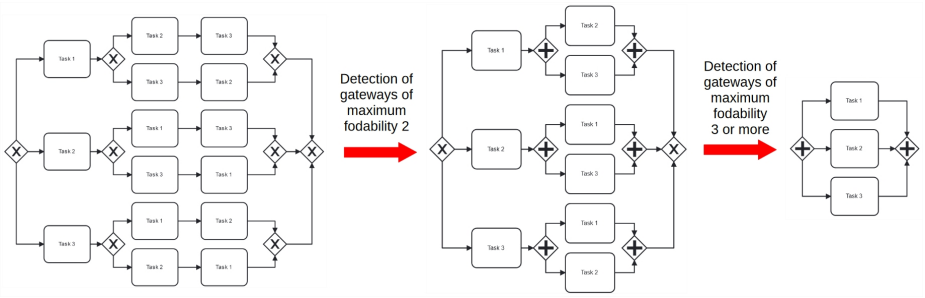

Fig. 10: Equivalence relationships during the folding process

in the unfolded BPMN process. Then, we return this BPMN process to the user. Figure 12 gives an overview of the whole folding process.

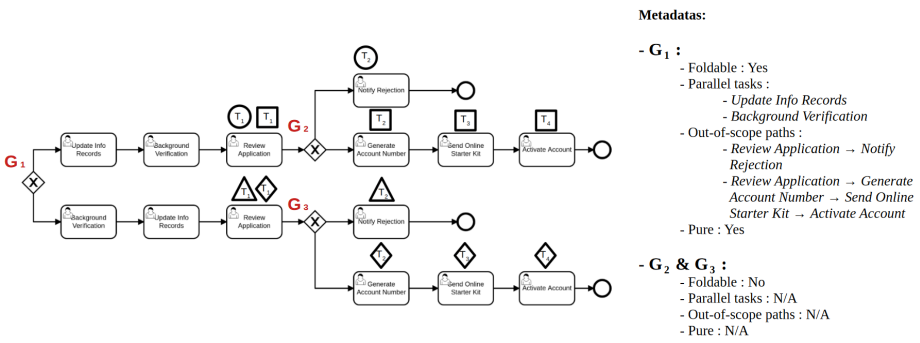

Fig. 11: Textual representation of the gateways' metadata

It is worth noticing that, starting from a BPMN process containing 24 tasks, the folding step returned a semantically equivalent process consisting of 7 tasks. In the end, we give back to the user a colored BPMN process, containing either black, green, or red tasks. From this process, the user knows that all the paths containing red tasks violate the given property, while paths containing green tasks satisfy it.

4 Tool and Experiments

4.1 Tool

The prototype tool implementing the approach was written in Java and consists of about 10,000 lines of code. It is worth noting that the prototype tool contains a lot of processing steps which have not been described previously in this paper, and which mostly consists in rewriting elements in other formats, such as BPMN

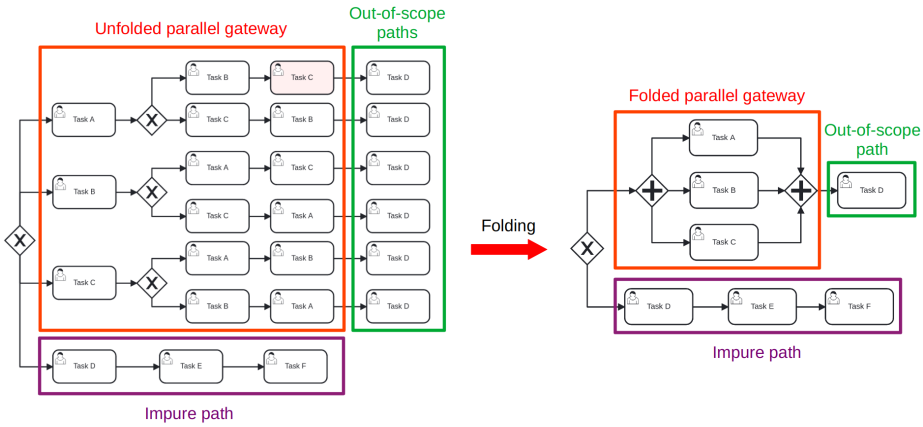

Fig. 12: Example of the whole folding process

to graph, BPMN to CLTS, CLTS to graph, etc. The tool allows the user to perform all the steps detailed in the previous section. It is worth noting that the tool has been tested and validated on about 150 BPMN processes.

4.2 Empirical Study

The goal of this study was to evaluate the usability of our approach in practice by quantifying the number of BPMN processes that are directly colorable, and if they are not, the number of processes that can be folded after the unfolding step. We used for this study 13 real-world processes taken from the literature, and corresponding to various application domains (such as transport, finance, or metallurgy). These processes have been given to the tool along with one well-suited safety property of the kind "T_1 *must not be executed before* T_2". The results of this study are the following: 21% of the BPMN processes were directly colorable, and 29% were succesfully folded after unfolding. Overall, in 50% of the cases, we were either able to color the initial BPMN process or to improve the unfolded one.

4.3 Performance Study

This study aimed at evaluating the performance of our approach following two different points of view: (i) assessing the scalability of the prototype tool over real-world examples, and (ii) evaluating the scalability of the prototype tool when increasing the size of the BPMN processes (in terms of nodes). Therefore, the BPMN processes used in this study are of three different kinds:

1. real-world examples of the empirical study presented in the previous section.
2. handcrafted examples that contain a high number of nodes.
3. handcrafted examples that are highly parallel.

The results of this performance study are presented in Table 1 and 2. Table 1 consists of three columns containing a short description of the BPMN process, the number of nodes in this process, and the execution time of the prototype tool.

BPMN Process	Number of nodes	Execution time of the prototype tool (s)
1. Vacations Booking	13	3.436
2. Account Opening	22	1.209
3. Publication Process	20	1.351
4. Steel Transformation	42	1.478
5. Plane Entry	27	2.235
6. Mortgage Application	15	1.974
7. Denoising	16	1.756
8. Credit Offer	15	1.004
9. Buying Process	29	1.540
10. Business Process	15	1.325
11. Job Hiring	29	1.798
12. Login Process	17	1.204
13. Support Ticket	22	1.431

Table 1: Results of the performance study on real-world BPMN processes

As far as real-world BPMN processes are concerned, the empirical study shows that the execution time is rather low. The highest time in the table is reached for the first process (vacations booking), for which the tool takes three seconds to execute all the steps of our approach. Performance of the tool is thus good for real-world BPMN processes. Table 2 presents the results of the study for larger examples. This table contains a first column containing the identifier of the BPMN process, followed by three columns presenting the sizes of the BPMN process and the LTS, and ends with four columns containing the execution times of each element of the toolchain and the global execution time.

As for the processes with many nodes (rows 1 to 4), one can see that the main source of computation time is due to the use of VBPMN and CLEAR, which take more than 99% of the execution time. As an example, this the case for process 4 in the table, which consists of hundreds of nodes, and for which our tool takes about 3 seconds to complete. Let us now focus on the examples with a high level of parallelism (rows 5 to 9). The unfolding phase of our approach generates processes with a high number of nodes (up to almost two hundreds of thousands nodes in the table for row 9). In such extreme situations, our prototype tool takes minutes or even hours to execute and complete its tasks. However, it is worth reminding that we have not found any real-world process with such a high level of parallelism. One may also mention that performing only partial unfolding would lead to a significant decrease of the execution time of the prototype on highly parallel processes. Nonetheless,

BPMN Process	Number of BPMN nodes	Number of states in the LTS	Number of nodes in the unfolded BPMN process	VBPMN execution time (s)	CLEAR execution time (s)	Prototype tool execution time (s)	Global execution time (s)
1. Generated Large 1	62	140	62	15.98	0.164	1.881	18.03
2. Generated Large 2	126	284	126	32.04	2.080	1.376	33.50
3. Generated Large 3	254	572	254	165.4	6.461	2.440	174.2
4. Generated Large 4	510	1,148	510	**902.3**	124.3	3.643	**1030**
5. Generated Parallel 1	19	529	11,582	13.07	0.086	19.76	32.92
6. Generated Parallel 2	20	606	23,165	13.15	0.089	82.68	95.92
7. Generated Parallel 3	21	683	43,614	13.23	0.088	466.8	480.1
8. Generated Parallel 4	22	770	87,229	13.61	0.082	**3948**	**3961**
9. Generated Parallel 5	23	857	165,307	17.81	0.122	**10100**	**10117**

Table 2: Results of the performance study on two types of handcrafted BPMN processes

the proposed approach is based on model checking techniques that perform a full unfolding of the BPMN process in order to verify the given property. As our starting point is the output of the model checker, we can not perform this partial unfolding.

5 Related Work

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 [21, 12, 13, 27, 11]. 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 [15], 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. Rewriting logic is also used in [14] 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.

Let us now concentrate on those using process algebras for formalizing and verifying BPMN processes, which are closer to the approach proposed in this paper. The authors of [29] present a formal semantics for BPMN by encoding it into the CSP process algebra. They show in [30] how this semantic model can be used to verify compatibility between business participants in a collaboration. This work was extended in [31] to propose a timed semantics of BPMN with delays. [8, 23] focus on the semantics proposed in [29, 31] and propose an automated transformation from BPMN to timed CSP. In [16] the authors have proposed a first transformation from BPMN to LNT, targeted at checking the realizability of a BPMN choreography. In [11], 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.

The approach presented in [19] proposes verification and comparison techniques based on model checking. The BPMN process is translated into LNT, which is then mapped to LTS before being given as input to a model checker. 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 [17] and [24] are also making use of model checking to perform verifications of temporal logic properties over BPMN processes.

All the approaches presented beforehand in this section are close to what we propose since they focus on the verification of BPMN processes. However, none of these works provide any solution to support the debugging of BPMN processes or to visualize counterexamples at the BPMN level.

Finally, we present the approach proposed in [28], which is the closest to ours. This approach aims at representing visually violations of temporal logic properties regarding a given BPMN process. The authors focus on what they called *containment checking*. Containment checking consists of verifying properties built from high-level BPMN processes (*e.g.*, representations of BPMN processes without many details) over low-level models (*e.g.*, complete representation of BPMN processes). In this work, the high-level BPMN process is translated into Linear Temporal Logic properties [20] while the low-level model is translated into SMV [7]. Both are given as input to the nuSMV [10] 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. Both works differ in terms of verification: model checking of safety properties for us whereas [28] performs conformance verification of a model regarding a higher-level model. Moreover, they visualize the counterexample only if it can be represented on the initial BPMN process (*i.e.*, if a matching exists), while we go beyond this case by providing additional coloration solutions if a matching does not exist.

6 Concluding Remarks

In this paper, we have proposed a way of improving and simplifying the comprehension and the visualization of safety property violations. The main objectives of the approach

were (i) to give a visual feedback of the violation expressed in BPMN notation, (ii) to stay as syntactically close as possible to the initial BPMN process while remaining semantically equivalent, and (iii) to be as minimal as possible in terms of BPMN nodes in the final process. To reach these goals, we chose the coloration of BPMN processes as visualization technique. More precisely, we have first presented a solution in which the original BPMN process is directly colored according to the satisfaction/violation of the property given as input. 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 applicable. Therefore, we have proposed a complementary approach which, in a first step, generates a new BPMN process from the CLTS (*unfolding*), and in a second step, performs some minimization over it to lower the number of nodes (*folding*). The different steps of the approach presented in this paper were implemented in a tool written in Java consisting of about 10,000 lines of code. In order to evaluate the usability and the scalability of this prototype tool, we performed two studies: an empirical one to get insights on 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.

The main perspective of this work is to support liveness properties. In this case, counterexamples and counterexample LTSs have different shapes (lassos) [4], and the approach thus deserves to be revisited to take this specificity into account.

References

1. Information technology - Object Management Group Business Process Model and Notation. 2013.
2. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
3. G. Barbon, V. Leroy, and G. Salaün. Debugging of Concurrent Systems Using Counterexample Analysis. In *Proc. of FSEN'17*, volume 10522 of *LNCS*, pages 20–34. Springer, 2017.
4. G. Barbon, V. Leroy, and G. Salaün. Counterexample Simplification for Liveness Property Violation. In *Proc. of SEFM'18*, volume 10886 of *LNCS*, pages 173–188. Springer, 2018.
5. G. Barbon, V. Leroy, and G. Salaün. Debugging of Behavioural Models with CLEAR. In *Proc. of TACAS'19*, volume 11427 of *LNCS*, pages 386–392. Springer, 2019.
6. G. Barbon, V. Leroy, and G. Salaün. Debugging of behavioural models using counterexample analysis. *IEEE Trans. Software Eng.*, 47(6):1184–1197, 2021.
7. B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. Mckenzie. *SMV — Symbolic Model Checking*, pages 131–138. Springer, 2001.
8. M. I. Capel and L. E. M. Morales. Automating the Transformation from BPMN Models to CSP+T Specifications. In *Proc. of SEW*, pages 100–109. IEEE, 2012.
9. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, C. McKinty, V. Powazny, W. Serwe, and G. Smeding. In *Reference Manual of the LNT to LOTOS Translator*, 2005.
10. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new symbolic model checker. In *International Journal on Software Tools for Technology Transfer*, pages 410–425, 2000.
11. 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.

12. G. Decker and M. Weske. Interaction-centric Modeling of Process Choreographies. In *Information Systems*, volume 36, pages 292–312, 2011.
13. R. Dijkman, M. Dumas, and C. Ouyang. Semantics and Analysis of Business Process Models in BPMN. In *Inf. Softw. Technol.*, volume 50, pages 1281–1294. Butterworth-Heinemann, 2008.
14. F. Durán and G. Salaün. Verifying Timed BPMN Processes using Maude. In *Proc. of COORDINATION*, volume 10319 of *LNCS*, pages 219–236. Springer, 2017.
15. N. El-Saber and A. Boronat. BPMN Formalization and Verification using Maude. In *Proc. of BM-FA*, pages 1–8. ACM, 2014.
16. M. Güdemann, P. Poizat, G. Salaün, and L. Ye. VerChor: A Framework for the Design and Verification of Choreographies. In *IEEE Trans. Services Computing*, volume 9, pages 647–660, 2016.
17. O. Kherbouche, A. Ahmad, and H. Basson. Using model checking to control the structural errors in BPMN models. In *IEEE International Conference on Research Challenges in Information Science (RCIS)*, volume 7, 2013.
18. A. Krishna, P. Poizat, and S. Gwen. Checking Business Process Evolution. In *Science of Computer Programming*, pages 1–26. Elsevier, 2019.
19. A. Krishna, P. Poizat, and G. Salaün. VBPMN: Automated Verification of BPMN Processes. In *Proc. of IFM'17*, volume 10510 of *LNCS*, pages 323–331. Springer, 2017.
20. F. Kröger and S. Merz. *Temporal Logic and State Systems*. Springer Berlin, Heidelberg.
21. A. Martens. Analyzing Web Service Based Business Processes. In *Proc. of FASE'05*, volume 3442 of *LNCS*, pages 19–33. Springer, 2005.
22. R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of FM'08*, volume 5014 of *LNCS*, pages 148–164. Springer, 2008.
23. L. Mendoza-Morales, M. Capel, and M. Pérez. Conceptual Framework for Business Processes Compositional Verification. In *Inf. & Sw. Techn.*, volume 54, pages 149–161, 2012.
24. T. Messaoud Maarouk, M. El Habib Souidi, and N. Hoggas. Formalization and Model Checking of BPMN Collaboration Diagrams with DD-LOTOS. In *Computing and Informatics*, volume 40, 2021.
25. R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.
26. Q. Nivon. Model Checking and Debugging of BPMN Processes using Coloration techniques. Master Thesis. 2022.
27. 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.
28. F. UL Muram, H. Tran, and Z. Uwe. Counterexample Analysis for Supporting Containment Checking of Business Process Model. In *International Workshop on Process Engineering (IWPE)*, volume 1, 2015.
29. P. Wong and J. Gibbons. A Process Semantics for BPMN. In *Proc. of ICFEM'08*, pages 355–374, 2008.
30. P. Wong and J. Gibbons. Verifying Business Process Compatibility. In *Proc. of QSIC'08*, pages 126–131, 2008.
31. P. Y. H. Wong and J. Gibbons. A Relative Timed Semantics for BPMN. In *Electr. Notes Theor. Comput. Sci.*, volume 229, pages 59–75, 2009.