UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Vostan Azatyan

On the transformation of Petri nets into BPMN models

Master's Thesis (30 ECTS)

Supervisor: Luciano García-Bañuelos, PhD

Tartu 2017

On the transformation of Petri nets into BPMN models

Abstract:
This thesis addresses the problem of translating a Petri net into an equivalent BPMN process model. This is fundamental problem with implications on the understanding of the semantics of the notation and that has potential applications in areas such process model discovery from event logs and structuring of process models. In previous work, it has been shown that the well-known family of free-choice Petri nets can be bidirectionally mapped into the subset of BPMN process models constructed solely with tasks and exclusive/parallel getaways. In contrast, this work searches at lifting the restriction to a larger family of Petri nets by proposing a translation that covers also the case of nets with symmetric confusion. The approach has been implemented in a prototype which has allowed us to conduct a preliminary performance study.

Keywords:
Business modeling, model to model transformation, Petri nets, concurrency theory
CERCS:P170, Computer science, numerical analysis, systems, control

Petri võrkude teisendamine BPMN mudeliteks

Märksõnad:
ärimodelleerimine, mudelilt mudelile teisendamine, Petri võrgud, konkurentsiteooria

Lühikokkuvõte:
Antud magistritöö käsitleb Petri võrkude teisendamist samaväärseteks BPMN mudeliteks. Täpsemalt öeldes keskendub see Petri võrkude alamklassile nimega töövoo võrgud. Antud lõputöös implementeeriti teisendaja, kasutades selleks mitmeid tehnikaid nagu näiteks Petri võrkude lahti pakkimine ja modulaarsed dekompositsiooni puud. Sellest tulenevalt pakub antud magistritöö välja täieliku teisendusalgoritmi, mis suudab käsitleda sümmeetrilisi segadusi Petri võrkudes. See on antud valdkonnas üks esimesi teisendamise meetodeid, mis katab ka seda klassi. Hetkel oleme teadlikud ainult ühest teosest mis illustreerib mõlemasuunalist teisendamist töövoo võrkude ja graafide vahel. Lisaks, esitleme me käitumissõltuvuste maatriksi arvutamise meetodi. Käsitleme ka erijuhtumeid, kus peame BPMN-is lisama tau sündmuse, et tegemist oleks samakujulise mudeliga.

Contents

1 Introduction **5**
1.1 Organization . . . 7

2 Background **8**
2.1 Petri nets and Workflow nets . . . 8
2.2 BPMN . . . 11
2.3 On the translation of BPMN and Workflow nets . . . 13
2.3.1 Bidirectional translation of BPMN and Petri net . . . 13
2.3.2 Related work . . . 17
2.3.3 Discussion . . . 18
2.4 Modular decomposition . . . 19
2.5 Unfolding of Petri nets . . . 20

3 Contribution **23**
3.1 The algorithm of translation . . . 23
3.1.1 Computing relation behaviour matrix . . . 24
3.1.2 First pass . . . 25
3.1.3 Second pass . . . 28
3.1.4 Third pass . . . 31
3.1.5 Forth Pass . . . 33
3.2 Proceeding the MDTs . . . 35
3.3 Adding tau transition . . . 37

4 Comparison and Results **39**
4.1 Dataset . . . 39
4.2 Performance . . . 40

5 Conclusion **42**

Licence 45

Chapter 1

Introduction

Nowadays, there are variety of different techniques, methods, and tools for analyzing the business processes. This explains the fact of existence of many different modeling notations. Hence, this generates the necessity of translation between these notations. This becomes especially evident, when the time comes for business analysts to take over as they prefer visual and more human readable high-level process models, with notations such as BPMN, UML activity diagrams and EPC over textual descriptions. However, several tools and techniques for model analysis use Petri nets as input. This is the case of tools for performance analysis, process mining and cost estimation.

The translation from high-level process models to Petri nets is well understood and there exist several solid translation methods in the literature [2, 12, 7].

Surprisingly, the translation from Petri nets to high-level process models has not had the same amount of attention. The only exception is probably the work reported in [15] and a few other papers on old model notations/languages [15]. However, we consider that this translation would also be valuable. For instance, many process mining tools currently generate Petri nets and not high-level process models. The reason seems to be that having a Petri net is good enough when you are planning to perform further analysis on the model and log (e.g. performance analysis). However, it is also interesting to have means to produce a high-level process model for analysts.

As a starting point and inspiration for this thesis was the paper [5] from Cédric Favre and Dirk Fahland and Hagen Völzer. In this paper they represent bidirectional translation method from *workflow graphs* to *workflow nets.* Though the algorithm is well formed and theoretically looks solid, we are not aware of any implementation of it. Thus, in the first place, we did implement this method, which as an input takes *free-choice, workflow net* and returns generated BPMN. In fact, the application performs well by returning reasonable output.

With this experience at hand, we decided to explore ways to lift the restriction of the method and cover other classes of Petri nets.

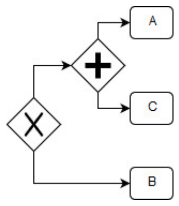


Figure 1.1: The BPMN example of confusion

In particular, we were interested in the family of Petri nets with symmetric confusion. This family of nets is characterized by including clusters of transitions where a subset of them are in conflict and are themselves concurrent with another subset of them with a topology similar to the one illustrated in figure 1.2. Note that this net is not longer a free-choice net, but can be the net produced by a process discovery technique on the log generated by the process model in figure 1.1.

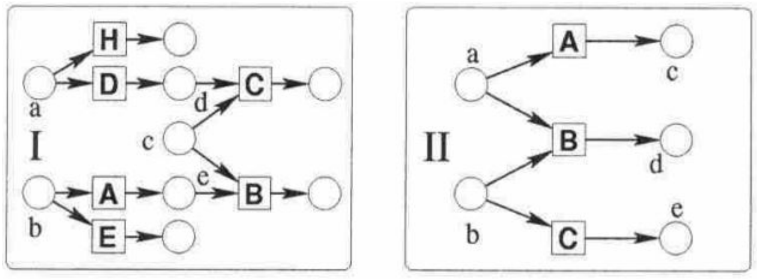


Figure 1.2: Confusions in Petr nets [6].

In general, there exist two types of confusion in Petri nets, which are illustrated in the Figure 1.2 (see [6]). The one in the left-hand side is referred to as *asymmetric confusion* and the other one is referred to as *symmetric confusion.* In this paper, we present algorithms for transforming Petri nets with symmetric confusion (thus including free-choice nets).

Finally, we note that our approach targets specifically the BPMN notation. The choose this notation because it is nowadays one of the most representative mainstream notations. However, we believe that the results can be reused with other notations such as UML activity diagrams and EPCs.

1.1 Organization

The rest of this document is organized as follows. In Chapter 2 we introduce the basic concepts on Petri nets, process model notations, and review the existing work on transformation from process models to Petri nets and vice-versa. In Chapter 3 presents our translation methods In Chapter 4 we report some performance related measurements. We conclude and discuss some directions for future work in Chapter 5.

Chapter 2

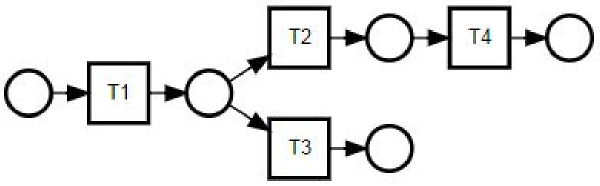
Background

2.1 Petri nets and Workflow nets

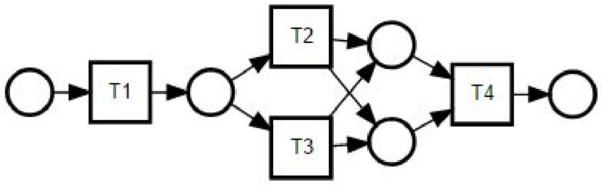
A Petri net is a mathematical formalism that serves as a foundation for the analysis and design of concurrent systems. Intuitively, a Petri net is a bipartite graph, with two types of nodes called Transitions and Places, all connected with arcs. Formally, we say that a Petri net is a triple $N = (P, T, F)$, where P refers to finite set of Places which represents the state of the process, T refers to finite set of Transitions those are to show the certain process execution and F stands for Arcs $F \subseteq (P \times T) \cup (T \times P)$ which are connecting Places to Transitions or backwards. Moreover, there can not be an arc between two Transitions or two Places. In the Figure 2.1a you can see an example of Petri net where $\{A, B\} \in T$ and $|P| = 4$. The *marking* of Petri net refers the state of the net during the process execution, formally the i marking of the net is the status of Petri net, meaning how many tokens each place holds in that particular moment of execution.

Every $x \subset P \cup T$ node in the net, will have $\bullet x$ for the *predecessors* of x so $\bullet x = \{y|(y, x) \in F\}$ and $x\bullet$ for the *successors* of node where $x\bullet = \{y|(x, y) \in F\}$. Given a set of nodes X (either transitions or places), w write $X\bullet$ to denote the union of the set of successors of each node $x \in X$, i.e. $X\bullet = \cup_{x \in X} x\bullet$. For instance in the figure 2.1b, the $A\bullet =< P > p1, p2$ where $A \in T$ and $(A\bullet)\bullet =< T > B$.

Workflow net is one of the subclasses of Petri net, with some structural and behavioral restriction. The Petri nets which are defining workflow processes are known as workflow nets [13]. Firstly, a workflow net must have one single source place (token in this place is presenting the case which going to be handled), referred to the start place, and one single sink place (the token in this place shows already handled case), also known as the end place. As well as every other node in a workflow net is reachable from the source place. See Figure 2.1b. Last but not least, formally $N = (P, T, F)$ Petri net will call workflow net if and only if [13], $\{source, sink\} \in N$ where $\bullet source = \emptyset$ & $sink\bullet = \emptyset$. These are the minimal requirements for Petri net to be workflow net which means that it is not necessarily to be workflow net if it has these requirements.



(a) Petri net



(b) Workflow net

Figure 2.1: Example of Petri net and workflow net

In addition to the aforementioned structural restrictions, a workflow net must comply with properties of proper completion(Liveness) and boundedness, conjointly referred to as soundness [14].

- **Soundness.**
  If a process starts with i amount of markings in the *source* place than in the end of the process it should be the i amount of tokens in the *sink* place[13]. In other words, there must not be any remaining token in the net at the end of the process. Additionally Workflow net soundness is required that all the Transitions in the Net could fire, for each T in the net there is a reachable state.

- **Boundedness.**
  The Workflow net said to be bounded if it does not contain more than

k tokens in each reachable marking from any place of the net. It also includes the initial place [8]. At the same time workflow net called $safe$ bounded if $k = 1$.

- **Liveness.**
  The liveness of workflow net refers to the not existence of deadlocks in the net[8]. Simply put, in the WN should not be possible to reache situation where there is T which can not be fired any further. This means that the workflow net's liveness is the guarantees of a deadlock-free net and not important which firing path was chosen.

  Workflow net has several degrees of liveness $L_0 - L_4$ [13].

  - L_0 **- live** when T will not be fired in any sequence of execution (Dead).
  - L_1 **- live** has a potential to be fired, there is T which is able to fire at least once in some execution sequence.
  - L_2 **- live** in case of given any k, T is able to fire k times in certain execution flow.
  - L_3 **- live** T can exist infinitely in different firing sequences.
  - L_4 **- live** means that L_1 - live for every marking in Workflow net.

The Free-choice nets (subclass of Petri net) usually are meant to represent workflow choices in nets, e.g. there are some Transitions in the Petri net which are sharing the same place so they represent OR-split construct. At the same time, the amount of the $predecessors$ has to be the same ($\bullet t_1 = \bullet t_2$) [13].

To put it formally, let's start from mathematical representation of free-choice nets $\forall p \in P : (|p^\bullet|) \vee (^\bullet(p^\bullet) = \{p\})$, meaning that every single arc from p to Transition in the Petri net is the only arc from place or the only one to Transition.

Free-choice nets have been trendy topic to study in the last decade because they have a huge capability to be analyzed, but, at the same time, they do not lose their expressive power [13].

2.2 BPMN

Business Process Model and Notation is the standard notation to capture business processes [9]. It is highly used in analyzing domains as well as designing high-level systems. Over the time, BPMN has been developed and currently inherits elements from several previously used notations, including the XML Process Definition Language (XPDL) [11] and the UML Activity Diagrams [10]. BPMN composed of many different elements which makes it a powerful and flexible tool for visualizing processes. The notation elements in BPMN can be separated into 5 groups.

- **EVENTS**
  Events represent an action resulting the flow of process and can subsequently be divided into start, intermediate and end events. Start event represents the start point of the sequence (process) or start of the message (message start) where the end of the process is called end event as well as there is end message event which accordingly shows the message end. Moreover, there are several intermediate events such as Message, timer and error that are to show the sending or receiving messages to/from other systems, the timer to put certain time interval during process and errors to show the faults and/or exceptions during the process execution. Those events are not only ones in BPMN, there are several other events which we are not going to describe in the context of this paper.

- **ACTIVITY**
  Activities can be subdivided into two types *Tasks* and *Subprocesses.* A *Task* is the core element for BPMN and represents activity, process or work to be performed (the same concept as *Transition* in pertinent). There are several event types such as *service, send, user, manual, script, receive* and *reference. Sub-processes* as the word itself defines it is the compound activity of one part of the process, in other words, it is the black boxed process. There are two types of *subprocesses*, *embedded* and *independent*. Embedded ones they are considered to be a part of the main process where the independent ones can be used in any other models. Activities also have other behavioral attributes such as *looping* or *multiple instances* [2].

- **SEQUENCE FLOW**
  Normal flow shows the direction of the process execution. The exceptional event fired in the case of some errors or time activity is happening, it is represented by attached *error* or *timer* event to the boundary of an activity. During the execution of normal flow, the error or time flows never will be executed. They will appear only in some exceptional case.

- **GATEWAY**
  Getaways are meant to specify points where a flow is divided into multiple alternative or concurrent flows inside the process. In general, a gateway can be divided into multiple subsequent paths. That type of gateway is also known as a split gateway. Similarly, a joint gateway is used whenever multiple paths converge into it. Parallel split gateways, also known as AND-split is to show the sequence (concurrent) flows [2], and for synchronizing this concurrency it used *parallel join* gateways $(AND - join)$. XOR to select one of many possible flows based on some external event or certain process data. If the XOR is data-based than it is called $XOR - split$ otherwise it is event-based and known as $deferredchoice$. There is another type of XOR getaway which is *marge getaway (XOR-join)* that joins the incoming flows. This implies that no matter which of the incoming flows have been triggered, it will continue the flow. Next is the *decision gateway (OR-split)* to chose one of the multiple outgoing flows. There is always receive or intermediate task preceded by *Event-based XOR decision getaway* to show the exact moment of execution.

- **MESSAGE FLOW**
  The $Message\ flows$ are meant to show the streams between two relative processes with some communication tasks e.g. send, receive task or message events. Two separate processes can be connected with $message\ flows$ which will be called $interacting\ processes$.

Overall, we are going to focus on basic BPMN constructs (*Tasks*, *start and end* events, *XOR* and *AND* gateways and only *normal flows*) on the scope of this paper. The other BPMN elements such as *sub-processes*, *pool/lanes* among others would be ignored.

In the figure 2.2 we can see the visual representation of the above mentioned elements.

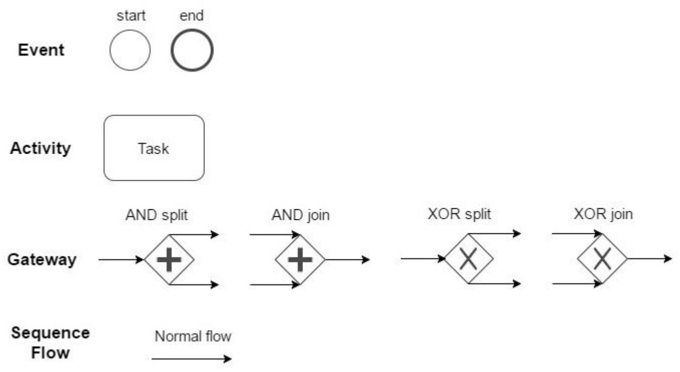


Figure 2.2: Basic BPMN constructs

2.3 On the translation of BPMN and Workflow nets

The Petri nets were investigated from the theoretical point of view for many decades and that is the fair explanation to the fact that many process analysis techniques/tools do support Petri nets such as Woflan, Woped, PROM among others.

To be more precise, Petri nets is quite technical for business analysts to understand and read them. At the same time, BPMN is a better candidate to visualize more complicated processes, which is better looking and more descriptive for business people. In fact, this is the main issue which pushes the science to think about translation between those notations.

The bulk of the work on the translation between BPMN and Petri nets can be divided into two large groups. First, we have the methods that translate the models from BPMN into Petri nets. Second, few other methods exist that translate models from Petri nets into BPMN. We are aware of only one work that does the translation in both directions. In this section, we are going to talk about several methods of the current state of translation techniques.

2.3.1 Bidirectional translation of BPMN and Petri net

As a starting point for our work, we took the paper [5] which, surprisingly is the only work we are aware of that describes a sound method for translating an input Petri net into a BPMN model.

Other papers exist that describe explicitly the translation in one of the directions, and in some cases, the translation is covered as a byproduct. In contrast, the work [5] provides a well-grounded theory for the translation and does it in the both directions. That is the reason why we look at the work of [5] deeper and we leave the discussion of the other works for later on in the Subsection 2.3.2.

In the context of this paper, they put the issue on a relationship between BPMN, EPC and UML activity diagrams as a *Workflow Graphs* (future in the paper W) and Workflow Nets (future in the paper N). Nevertheless, in the article there is a solution for both way translation, and all work is based on the bidirectional mapping. Yet, we are going to concentrate more on the translation from N to W.

First of all, to translate N to W or backward will require both of them to be normalized, limited in length and complexity. The *Workflow Graph W* or *Petri Net N* called normalized if any x element of the process is normal. The x will be called normal if and only if it has at most one incoming and one outgoing flow. The translation done by the simple mapping of x normal elements, shown in the Figure 2.3.

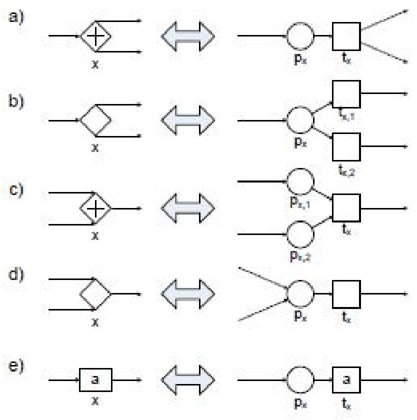


Figure 2.3: Mapping between Workflow Nets and Workflow Graphs [5]

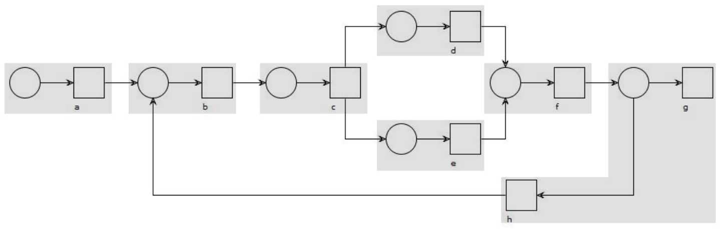
The patterns in the Figure 2.3 are used to translate from N to W, in this case, they introduce the way to group *Workflow Net* elements onto *conflict clusters*. The article claims that the every normalized Petri Net is able to decompose into *conflict clusters*. The way of breaking the Petri nets onto conflict clusters represented in the article is simple and straightforward, see the Algorithm 1. More precise, transformation implementation will be done in 4 steps [5]:

- **Get *conflict clusters***
  From Workflow net N get set of *conflict clusters* (P_1, T_1, F_1), ... , (P_n, T_n, F_n), using the Algorithm 1.

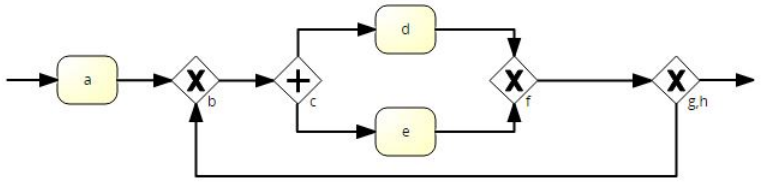
- **Find pattern**
  Identify pattern according to right hand side of Figure 2.3, add accordingly left hand side *node* to W output. Adjust all arcs in pattern accordingly.

- **Connect all elements in *W***
  The next step is to connect all *nodes* in W output, by mapping back to N.

- **Add *source and sink***
  The last step is to add source incoming edge and sink outgoing edge, which simply done by identifying the *source* palace in N and add an incoming edge for according *node* in W same done for *sink* place.



(a) Petri net with identified patterns highlighted



(b) Output BPMN model

Figure 2.4: Translation example

For instance, to illustrate the algorithm as an example we took Figure 2.4a, there is highlighted the identified clusters by the Algorithm 1. Then by iterating over clusters it identifies correct patterns and translates onto BPMN out of this clusters, the result is illustrated in Figure 2.4b.

In this figures, we easily can see the mapping between those clusters and identified patterns. Moreover, from here it simply can be noticed that both models are isomorphic which is prove that translation was correct.

Algorithm 1 Getting *conflict clusters* from Petri Net

```
$cluster \leftarrow \emptyset$
$workset \leftarrow P$

while $workset \neq \emptyset$ do
    let $p \in workset$
    places $\leftarrow \emptyset$
    transitions $\leftarrow p\bullet$

    for each $t \in transitions$ do
        places $\leftarrow$ places $\bigcup \bullet t$
    end for

    $cluster \leftarrow cluster \bigcup \{< places, transitions >\}$
    $workset \leftarrow workset \setminus places$
end while
```

In addition, the article represents some methods to normalize the Petri nets by adding some additional transitions to the net. For example, if we have a look to Figure 2.4a and assume we do not have transition b than it is noticeable that there will be an incomplete cluster and the translation will fail. For this reason, in the paper, they propose to add this transitions in order to have complete clusters.

To conclude, the translation method represented by [5] is quite narrowed and covers a small class of Petri nets only. To highlight, the method is able to translate only and only *free-choice* Workflow nets which have.

The goal of this thesis is to extend this method and broaden the class coverage of Petri nets.

2.3.2 Related work

In the following subsection we discuss representative work in the field, organized according to the direction of the translation.

From High-Level Process model to Petri nets

In the article of Remco M. Dijkman, Marlon Dumas and Chun Ouyang [2] the method of translation from BPMN to Petri net is described. The translation presented in flowing way, for each possible BPMN object and group of objects there exists corresponding mapping in Petri net. First of all to translate BPMN to Petri net it has to be *well-formed*. To be more precise: 1) The start event should have one outgoing flow and non incoming. 2) End event has only one incoming flow and none outgoing one. (To point out, these two restrictions are coming with actual BPMN syntax). 3) In all over BPMN the activities and intermediate events have only one incoming and outgoing flow. 4) $AND - split$ and $OR - split$ getaways have one incoming and and many outgoing flows. 5) Similarly the $AND - join$ and $OR - join$ getaways have one outgoing and many incoming flows. In fact, all 3 rules that we defined above are achievable in any BPMN model by applying some additional mapping. To finalize, the paper proved that almost in all cases it will be possible to convert BPMN to Petri net.

The research reported in the series of papers Artem Polyvyanyy, Luciano Garcıa-Banuelos, Marlon Dumas [12], presents an approach to structuring acyclic process models. A structured process model is one where for every node with multiple outgoing arcs (split) there is a corresponding node with multiple incoming arcs (join), and vice versa, such that the fragment of the model between such nodes forms a single-entry-single-exit subgraph [12]. In that context, the research reported in [12] search a provide a complete method for transforming any input model (acyclic) into its structured version, if any exists. To that end, the method relies on a series of transformations, which preserve the behavioral equivalence of the input and output model. The input BPMN model is decomposed into a the fragments of the BPMN that are structurally unstructure is into a (free-choice) Petri net, which is then unfolded into an occurrence net. The occurrence net is then used for computing a behavior relation graph that summarises the set of behavior relations observed in the input Petri net.

One more journal [7] presents translation method from BPMN to Petri net. The translation method has the restrictions for input BPMN which has to have single start and end events. Also, several gateway types are not allowed, in particular OR-gateways, which represent the ‘Multi-choice’ and ‘General Synchronizing Merge’ patterns. In general the mapping base on the control-flow aspect of processes. When transforming a model, first, each object is mapped onto a partial Petri net and second, the partial Petri nets are composed into a complete model. Although this approach works for many constructs, some constructs cannot simply be mapped and then composed. The mapping from BPMN to workflow nets allows the soundness of these nets to be analyzed [7].

From Petri nets to High-Level Process model

One of the first publication about converting workflow nets into the hierarchical decomposition of specific BPEL [15]. BPEL (Business Process Execution Language) is built on top of XML and Web Services. The BPEL uses XML-based language, although it has graphical editors which are usually used by managers or business analysts. Hence, we are going to put BPEL parallel with BPMN, although BPEL is more technical and usually unreadable for business people due to complicated syntax, it supports many graphical tools, which is making it quite similar to BPMN. At the same time, there are both ways converters from BPMN to BPEL or vice versa. Accordingly, it will be quite reasonable to have a look this translation.

In the work, they present the algorithm which produces a BPEL specification from given Petri net in the input. They are one of the first who claimed that it will be more interesting to translate from Petri nets to BPEL rather than backward. However, as it is one of the first transformers to this direction, it has lots of restrictions and gaps to be accurate, meaning it only can translate a narrow group of workflow nets only.

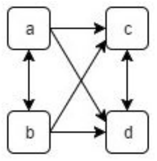
2.3.3 Discussion

Interestingly, there is a small number of articles describing the opposite translation. There have been few researches done about translating Petri nets into BPM graphs. Though, it would be a really big commitment to have such a tool which will transform to this direction. Many business analysts will get a huge benefit out of it, they could simply convert Petri nets and get some intuition about business workflows.

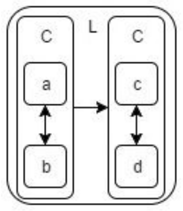
To conclude, in some cases, there is strong need to translate from already analyzed Petri net to BPMN in order to visualize it and in certain cases there is a need to translate vice versa in order to analyze the process. AT the end of the day, bidirectional translation is really important in the spec of analyzing the processes.

2.4 Modular decomposition

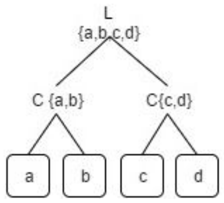
In this section, we will discuss a part of the graph theory and the concept of decomposition of the graph into the modules.



(a) Simple graph



(b) Modular decomposition of the graph



(c) MDT of the graph

Figure 2.5: Example of modular decomposition

The *modular decomposition* is the reformation of the graph to the subset of modules [16]. A module is a type of representation of connected elements in the graph. For example, modules can be constructed recursively, as so that each *module* can hold others. Hence, the Modular decomposition tree (MDT) is the tree with decomposed *modules*. For instance, in the figure 2.5 we can see the simple modular decomposition with corresponding *decomposition tree*.

The MDT can be defined as follows [12]: Let $G = \{V, E, L\}$ be a graph where V is vertexes, E edges and L is the set of labels. The $MDT(G)$ will be denoted as the function generating MDT given graph. For instance in the figure 2.5c shown the MDT of the 2.5a graph. However, if we give the textual representation of the MDT on the figure 2.5c it will look like $L[C[a, b], C[c, d]]$.

Let us classify the modules M of graphs on the scope of this paper.

- COMPLETE (C) : This is the case where we have one complete path of the graph. Formally, M module is $COMPLETE$ iff exist $l \in \{\#, \|\}$

and all vertexes in module have a same relations (let $x, y \in M$ and $E(x, y) = l$). This module can be of two types, first, the case where vertexes in the graph are concurrent ($||$) will be COMPLETE_1 module. Second, where the vertexes in the module are in conflict we will call them COMPLETE_2.

- LINEAR (L) : If the elements in M has a linear order.
- TRIVIAL (T) : These are single cell M where there is no arc and it is the only element in the module.
- PRIMITIVE (P) : Iff$M$ non of the above classes than it is classified as an *primitive*.

Later in the section 3.2 we will talk about how to generate BPMN out of MDT.

2.5 Unfolding of Petri nets

Unfolding is the process of producing *Occurrence net* (Occurrence net is a Petri net that can have places with multiple outgoing edges but all places have at most one incoming edge) from a Petri net.

An *Occurrence net* $N = (B, E, F)$ (B is *conditions* E is *events* and F is the arcs between B and E) can be defined as follows [3]:

- For all $b \in B$ such that $| \bullet b| \leq 1$ and N is acyclic
- For every $e \in E \cup B$ the set $\{f \in B \cup E \mid (f, e) \in G^{+}\}$ is finit
- There is no $e \in E$ which is in self-conflict.

Algorithm 2 Unfolding Algorithm [3]

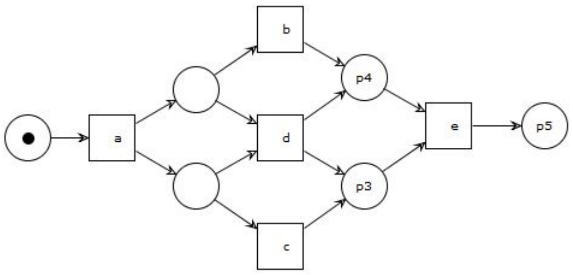
```
1: $input$ : is a Petri net with initial marking in the $source$ Place.
2: $output$ : is $U$ unfolded net.
3: $U \leftarrow (b_0 = (source, \emptyset))$
4: $pe \leftarrow getPE(U)$                ▷ returns the set of possible extensions in U
5: $cutoff \leftarrow \emptyset$
6: while $pe \neq \emptyset$ do
7:     let $event \in pe$
8:     if $[e] \bigcap cutoff = \emptyset$ then
9:         $U \leftarrow U \bigcup e$
10:        $U \leftarrow U \bigcup$ $conditions\ activated\ by\ current\ event$
11:        $pe \leftarrow getPE(U)$
12:        if $U.isCuttOff(e)$ then          ▷ checks if given $e$ is cutoff in U
13:            $cutoff \leftarrow cutoff \bigcup \{event\}$
14:        end if
15:    else
16:        $cutoff \leftarrow cuttof \setminus \{event\}$
17:    end if
18: end while
```

Net system is the Petri net with the particular marking. For instance, M_0 is initial marking of the Petri net where $p \in M_0 \Leftrightarrow M_0(p) =$ *amount of tokens*. To unfold Petri net there is several steps need to be applied [3]. Unfolded net $U = (E, B)$ where: $E = (t, B)$ stand for *events* which has two components transition and the list of conditions where event can be executed, followed by $B = (p, E)$ conditions with the place as a first element and list of events where the execution of event can put system to the following condition.

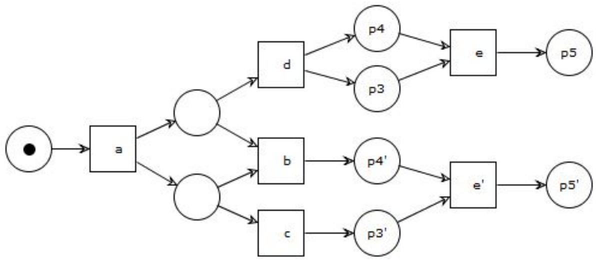
In the scope of this paper, we will assume that initial marking of the net is the marking where we have one token in the source place.

- U unfolded net should be initialized with all set of places which has token/s.
- get initial place with marking and make a condition with empty set of events and put it to U

If we apply this algorithm correctly we will end up with correct *Occurrence net*. In the Figure 2.6, is one example of unfolded Petri net by algorithm 2.



(a) Before unfolding



(b) After unfolding

Figure 2.6: Unfolding example of Petri net before/after.

Chapter 3

Contribution

3.1 The algorithm of translation

The translation of BPMN done in several passes during which the initial BPMN (the BPMN which was generated in the first pass) will be transformed onto correct translation.

However, before executing the passes we need to have *Behaviour Relation Matrix*, the explanation of computing the matrix will be given in the section 3.1.1.

- **Fist pass:** In the first pass, we generate the initial version of the BPMN model, where all events except for cutoff and duplicate events are represented. Note that this pass will give rise to an acyclic model, isomorphic to the unfolding and, hence, no merging XOR gateways are included.

- **Second pass:** In the second pass, we merge the future of the cutoff events. Note that a cutoff event happens to be a duplicate, then this cutoff will be processed only later in the third pass.

- **Third pass:** As mentioned, in this pass we are going to resolve all cutoff duplicates. BPMN will be modified accordingly, XOR joins will be added where relevant.

- **Forth pass:** In this pass we are going to take into account the copied events which are not cutoff, and do final changes on BPMN by adding *AND/XOR splits*.

In the end, we may end up with some not used elements in BPMN which will be cleaned up.

Besides, pay attention that input Petri net should be already unfolded by algorithm 2 introduced by [3].

Let us have an in depth look on each pass with examples.

3.1.1 Computing relation behaviour matrix

In the Petri net two transition, a and b are behaviourally related to each other. Generally speaking, they can be in $CONFLICT$, $CAUSALITY$ or $CONCURRENCY$.

- $CONFLICT$ The a and b are in conflict iff, the execution of each of them will disable another one, meaning if we fire a the b never can be fired anymore and vice versa.
- $CAUSALITY$ The a is in causality with c if the execution of a will enable c or there is exist such b which is enabling c and at the same time a in causality with b. This is called *transitive causality*. In other words it is known as sequentiality of processes.
- $CONCURRENCY$ The concurrency shows the independence of two transitions, for instance, if a will be fired it will not affect b and vice versa, that means that a and b are in concurrency. This usally refers to parallelism of two processes.

The behaviour relation matrix (BRM), is the matrix which shows direct and transitive relations between transitions in the Occurrence net resulted from unfolding.

To compute this matrix we used the algorithm provided by [12]. The algorithm is composed with two parts. First, it computes the ordering transitive relations of unfolded net. Second, updates the relations of transitions in the local configuration of every cutoff transition.

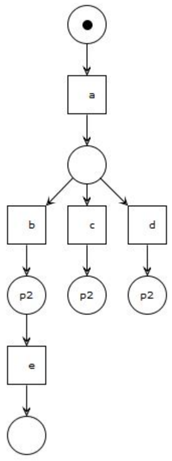
This matrix will be used as a key to compute directed relation graph for each cluster in every pass. In the first pass 3.1.2 we will present the example of already computed relation matrix.

3.1.2 First pass

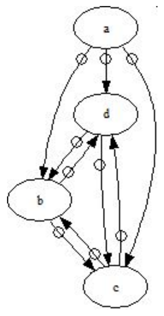
Algorithm 3 First pass

```
 1: $E' \leftarrow E \setminus$ copies
 2: open $\leftarrow E'$
 3: **while** open $\neq \emptyset$ **do**
 4:     Choose any event curr $\in$ open
 5:     Let $P, Q \subseteq E'$ be sets of events s.t. curr $\in$ P and P $<_\mu^m$ Q
 6:     open $\leftarrow$ open $\setminus$ P
 7:     $\mathcal{N} \leftarrow P \cup Q$
 8:     $\mathcal{D} \leftarrow (<_\mu \cap P \times Q)$                ▷ Direct causality relation
 9:     $\mathcal{C} \leftarrow (\# \cap \mathcal{N} \times \mathcal{N})$                ▷ Conflict relation
10:     mdt $\leftarrow$ COMPUTEMDT($\mathcal{N}, \mathcal{D} \cup \mathcal{C}$)
11:     CREATEANDADDPROCESSGRAPH(process, mdt)
12: **end while**
```

Above you can find the algorithm 3 which introduces the first pass of translation onto BPMN. The result is initial BPMN from behavior relation graph.



(a) Unfolded petri net without cutoff event



(b) Directed graph for first iteration

Figure 3.1: Example for first pass

Figure 3.1a is an example of input for above algorithm. Let us start with going over algorithm and apply it to the Petri net. As the first example, we decided to use a rather simple Petri net to avoid the complex cases. Hence, no event is duplicated and there are not loops in it. Note that the Petri net is not the original one rather already unfolded one.

First, we have that the set of events in the unfolding is $E = \{a, b, c, d, e\}$.

Moreover, the direct causality relation matrix (DCRM) 3.1 for our example is the following: $<_{\mu}{}' = \{(a, b), (a, c), (a, d), (b, e)\}$ and the conflict relation is: $\#' = \{(b, c)\ (c, b), (c, d),\ (d, c),\ (d, b),\ (b, d)\}$.

$$DCRM = \begin{pmatrix} & \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} & \mathbf{e} \\ \mathbf{a} & | & < & < & < & < \\ \mathbf{b} & . & | & \# & \# & < \\ \mathbf{c} & . & \# & | & \# & \# \\ \mathbf{d} & . & \# & \# & | & \# \\ \mathbf{e} & . & . & \# & \# & | \end{pmatrix} \tag{3.1}$$

In line 1, we make a copy of E such as we remove all the duplicate events. In this example, E' is identical to E.

Next, we copy E' into *open*, which corresponds to our working set. Hence, we will repeatedly execute the lines 3 to 12 as long as there are events in *open*.

When we enter the while loop in line 4, we select one event and keep it in *curr*. Note that the order in which the events are processed is not important, yet the result will be the same. Hence, let us assume we select event a. In the next line, we compute the sets P and Q such that, for our example, Q contains the set of direct successors of a and P contains the set of direct predecessors of the events in Q. Note that P and Q must be maximal. For instance, the sets $P' = \{a\}$ and $Q' = \{b, c\}$ are not consistent with the requirement of maximally, because Q' does not contain d. The notation $<_{\mu}^{m}$ refers to the set of maximal direct relations.

In the next step, from the *open* set we are going to exclude all P because we do not want to iterate over them twice.

With the sets P and Q, we will build a *behavioral relations graph* (BRG) as introduced in [12]. A BRG is a directed graph, where represent the three behavior relations (conflict, causality and concurrency) as follows: causality is represented with direct edges, conflict is represented with undirected edges (or edges in both directions) and concurrency is represented by the absence of edges. In [12] it is shown that this encoding enables the use of the modular decomposition, which then can be used to generate a BPMN fragment. The graph in Figure 3.1b corresponds to the BRG of the first iteration in our example, that is, to the behavior relations of the events $P \cup Q$.

To construct the BRG we proceed as follows. First, in line 7 we compute the set of nodes in the BRG, which in our example corresponds to $N = \{a, b, c, d\}$ Then, in Line 8 we compute the edges that represent the (direct) causality relation. To compute this relation, we take the global relation $<_\mu$ and we compute the restriction to the events that we are using, i.e. $P \times Q$. Thus, we have that $((P \times Q = \{(a, b), (a, c), (a, d)\}) \cap <_\mu = \{(a, b), (a, c), (a, d)\})$.

In line 9, we compute the conflict relation using a similar approach. As so, N^2 and $\#$ $((N^2 \cap \# = \{(b, c), (c, b), (c, d), (d, c), (d, b), (b, d)\})$.

Next we have the *computeMDT* function from [12] which as an argument will take N, D and C and will generate modular decomposition tree.

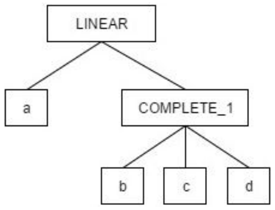


Figure 3.2: Decomposition tree visualization for first iteration

Finally, the decomposition tree for first iteration will look like *LINEAR[COMPLETE_1[c, b, d], a]*, visualization of tree will look like figure 3.2.

The last line in the algorithm is to add and create all corresponding elements in the decomposition tree, which will be discussed on section 3.2.

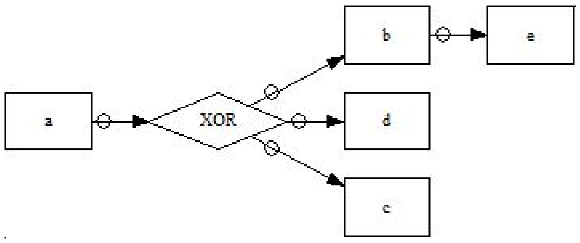


Figure 3.3: Inital BPMN, resuly of the algorithm 3

Finally, in the figure 3.3 we can see the initial BPMN result from completion of algorithm 3. As we see the XOR split point is generated but no

XOR join exists yet.

Also, pay attention that MDT in the figure 3.2 dose not contains e event but in the figure 3.3 it exists. This is because, the event e was generated during the second iteration of the first pass, which is not represented here.

3.1.3 Second pass

In this pass we are going to proceed with *cutoff* events, and add corresponding XOR join elements to initial BPMN from the first pass. Note, that some cutoff events can be duplicated in this case we leave them to proceed later on in 3d pass.

Algorithm 4 Second pass

```
1: $E' \leftarrow E \setminus \text{copies}$
2: $\text{open} \leftarrow \text{cutoffs} \setminus \text{copies}$
3: while $\text{open} \neq \emptyset$ do
4:     Choose any event $\text{curr} \in \text{open}$
5:     Let $P, Q \subseteq E'$ be sets of events s.t. $\text{corr}(\text{curr}) \in P$ and $P <_{\mu}^{m} Q$
6:     $P' \leftarrow P \cup \{e \mid e \in \text{cutoffs} \wedge \text{corr}(e) = \text{corr}(\text{curr})\}$
7:     $\text{open} \leftarrow \text{open} \setminus P'$
8:     $\mathcal{N} \leftarrow P' \cup Q$
                                          ▷ Translated direct causality relation
9:     $\mathcal{D} \leftarrow \{(e, f) \mid e \in P', f \in Q \wedge (e <_{\mu} f \vee \text{corr}(e) <_{\mu} f)\}$
10:    $\mathcal{C} \leftarrow \begin{cases} (e,f) \in \mathcal{N} \times \mathcal{N} & \text{if } e, f \notin \text{cutoffs} \wedge e \# f \\ (e,f) \in \mathcal{N} \times \mathcal{N} & \text{if } e \in \text{cutoffs} \wedge e \# f \wedge \neg(\text{corr}(e) <_{\mu} f) \\ & \wedge (\nexists g \in \mathit{copiesOf}(f) : e \parallel g) \end{cases}$
11:    $\text{mdt} \leftarrow \text{COMPUTEMDT}(\mathcal{N}, \mathcal{D} \cup \mathcal{C})$
12:    CREATEANDADDPROCESSGRAPH(process, mdt)
13: end while
```

Here we can see the algorithm 4 which represents the second pass, it is highly noticeable that it is similar to what we had in the first pass. Hence, to keep the example simple we will use same input from figure 3.1a and add missing XOR join.

For this reason, in the second line instead of taking $E\prime$ as a *open* set we will take all non-copied cutoff events. Next, we will execute from 3 line till 14 until the *open* set becomes empty.

In the example, the list of $cutoff$ events will be $\{c, d\}$, and the corresponding event for this two cutoff events is event b.

When while loop is executed similarly to first pass we select one event and keep it in $curr$. As the order of the iteration over the *open* set is not important, let us assume that c event was selected as a $curr$. Next line, we need to compute the sets P and Q such that, P must contain the corresponding event of $curr$ which is $\{b\}$ and Q will be the set of all direct successors of b.

In line 6 we need to extend P set with the all cutoff events which has the same corresponding event and new set we call $P\prime$. In the example $P\prime = \{b, c, d\}$ as we see that actually both c and d have the same corresponding and they both are cutoffs. Next, we will clear open from $P\prime$ events in order not to iterate over them again.

As in the first pass, now we need to compute N, D and C with a slightly different approach than in the first pass.

In line 8 we compute the N with the all set of the elements that are going to be in BRG, based on our example it refers to $N = P\prime \cup Q = \{b, c, d, e\}$.

Next, in line 9, D will be the set of tuples where the first element of the tuple is from $P\prime$ and second element is selected from Q in such a way that it will be direct successor of the $firstelement$ or direct successor of the corresponding event of $firstelement$. As we see in the example, $D = \{(b, e), (c, e), (d, e)\}$, note that in the input Petri net we do not have relation between $d- > e$ but the corresponding event b has direct relation with e.

Finally, on the line 10 we will compute C set which is comparably more complex than in the other cases. The conflicts will be computed from the intersection of two conflict sets. First set, when there is (e, f) which are not cutoff and the f and e are in conflict. Second, if there is e which is in cutoff and e in conflict with f but the corresponding of e not successor of f and there is no such g copy of f in the way that e and g are concurrent. The result, will be the set of $C = \{$*(b,c), (c,b), (d,b), (b,d), (d,c), (c,d), (d,e), (e,d), (c,e), (e,c)*$\} \setminus \{$*(e,d), (e,c), (e,b)*$\}$.

Now we are ready to generate BRG and then add corresponding XOR join to initial BPMN.

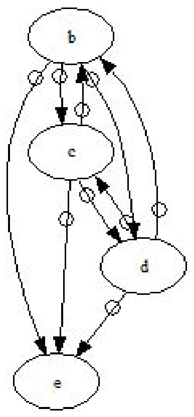


Figure 3.4: Directed graph for first iteration in second pass

From the second pass first iteration modular decomposition tree will represented as follows $LINEAR[COMPLETE\_1[b, c, d], e]$.

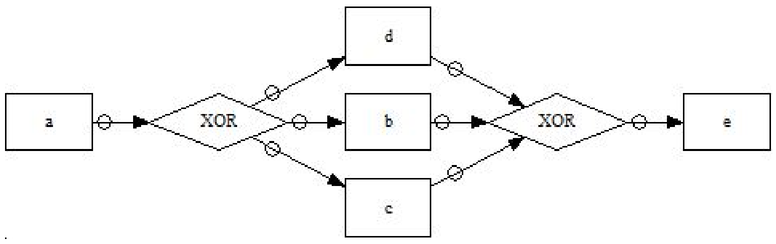


Figure 3.5: Generated BPMN after second pass, result of the algorithm 4

In the end of the second pass, we can see that the BPMN is constructed figure 3.5 and is isomorphic with initial Petri net. Hence, there are no future passes that will be interesting to review with this example as there are no copies which are not cutoff and there are no cutoffs which do not correspond to the same event.

3.1.4 Third pass

Algorithm 5 Third pass

```
1: open ← cutoff ∩ copies
2: while open ≠ ∅ do
3:     Choose any event curr ∈ open
4:     Let $I \leftarrow \{e \mid e \in \text{cutoffs} \wedge \text{corr}(curr) = \text{corr}(e)\}$
5:     Let $P, Q \subseteq E$ be sets of events s.t. $I \cup \{\text{corr}(curr)\} \subseteq Q$ and $P <_{\mu}^{m} Q$
6:     $Q' \leftarrow Q \setminus I$
7:     open ← open \ I
8:     $\mathcal{N} \leftarrow P \cup Q'$
                                              ▷ Translated direct causality relation
9:     $\mathcal{D} \leftarrow \{(e,f) \mid e \in P, f \in Q' \wedge e <_{\mu} f\} \cup \{(e, \text{corr}(f)) \mid e \in P, f \in I \wedge e <_{\mu} f\}$
10:    $\mathcal{C} \leftarrow (\# \cap \mathcal{N} \times \mathcal{N}) \setminus \mathcal{D}^{-1}$          ▷ Conflict relation
11:    mdt ← computeMDT($\mathcal{N}, \mathcal{D} \cup \mathcal{C}$)
12:    createAndAddProcessGraph(process, mdt)
13: end while
```

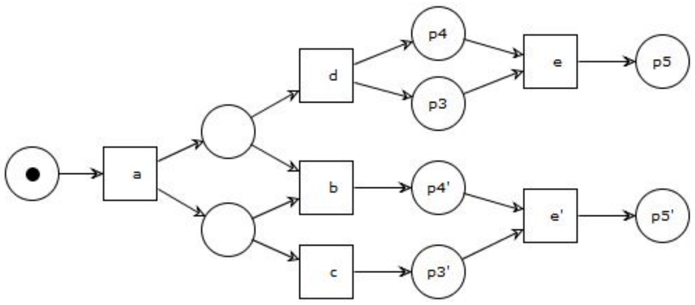


Figure 3.6: Unfolded Petri net with cutoff and copy

In this pass we will focus on all cutoff duplicate events, due to this we will use more complex example shown in figure 3.6.

We start to go ahead and execute algorithm 5 assuming that as an input given the above represented unfolded net.

In the first place, likewise the previous passes, we need to get the *open* workset. For this case, it will be chosen from the intersection of copies and cutoffs (all cutoffs which are the copy at the same time). As a result, we can see that the only cutoff is the *e'* which is duplicated at the same time so the *open* = $\{e\prime\}$. Note, that in this case $e\prime$ is the only element so accordingly $curr = e\prime$ and corresponding of it will be the e.

In line 4 we will select the set of events I in such a way that, selected event will be the cutoff and the corresponding event of it will be the same as the one of $curr$. In the example we can notice that it will be the event $e\prime$ which is the copy of e so $I = \{e\prime\}$.

Now we need to compute P and Q in the line 5, such that Q will be the union of corresponding of $curr$ and I. Similar to other examples, P will be all direct predecessors of the events in Q. For instance, $Q = \{e, e\prime\}$ and $P = \{b, c, d\}$.

In line 6 we will make a new subset of Q in such way that newly created $Q\prime$ will have all elements in Q excluded all in I, i.e. $Q\prime = \{e\}$. Next, as in the other cases, we remove all I events from the list of open so we will not iterate them again.

From the line 8-10, we compute N all local events $N = \{b, c, d, e\}$, D translated direct causality relations and C conflict relations.

To compute D in the line 9 we will take the union of two sets, where the first set will be the direct relations between P events to $Q\prime$ and the second set will be the direct relations from all P events to corresponding events in I, i.e. $D = \{(d, e), (b, e), (c, e)\}$.

Followed by, in line 10 we will compute C set of conflicts. First, we compute it as we did for the first pass $N^2 \cap \#$ and then we face the problem of c and b being in conflict with e. Though, it is not hard to notice in the example that they actually are not in conflict in the initial model. To solve this issue we are going to make mirroring set of D and exclude any possible intersections with the set of the conflicts.

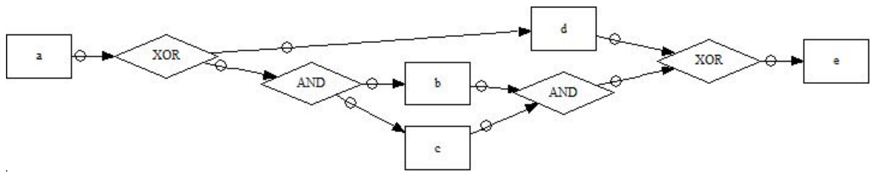


Figure 3.7: Generated BPMN after third pass, result of the algorithm 5

Likewise the other passes later MDT will be created from BRG. Finally, corresponding changes will be applied on the BPMN. After this pass, the BPMN will look like figure 3.7. Note that only join getaways have been added to the process model in this pass, the corresponding split gateways have been added in the first pass.

3.1.5 Forth Pass

In the previous passes, it is notable that none of the examples had a copy event which was not cut off. With the simple example, it would be impossible to demonstrate this case that way. That way we will show the smallest possible example when during the unfolding can apprise the event which is the copy but not cutoff.

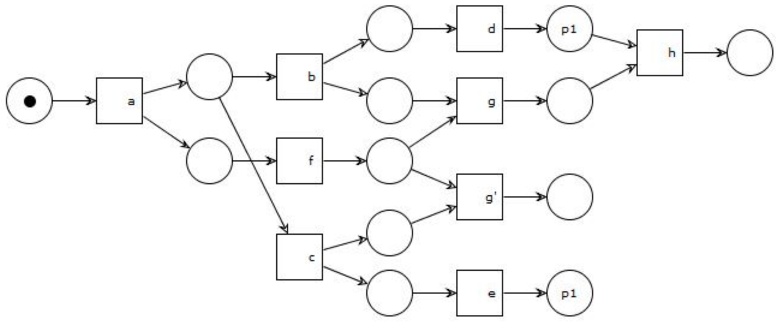


Figure 3.8: Unfolded Petri net with cutoff and non-cutoff copy

In the figure 3.8 we can see the example which is valid as an input of algorithm 6, the event $g\prime$ is the copy of g but at the same time it is not cutoff.

Algorithm 6 Forth pass

```
1: open ← copies \ cutoff
2: while open ≠ ∅ do
3:     Choose any event curr ∈ open
4:     Let $P, Q \subseteq E$ be sets of events s.t. $curr \in Q$, $\#(curr) \cap Q = \emptyset$ and $P <_\mu^m Q$
5:     open ← open \ {$curr$}
6:     $\mathcal{N} \leftarrow P \cup Q$
7:     $\mathcal{D} \leftarrow (<_\mu \cap P \times Q)$          ▷ Direct causality relation
8:     $\mathcal{C} \leftarrow (\# \cap \mathcal{N} \times \mathcal{N})$          ▷ Conflict relation
9:     mdt ← COMPUTEMDT($\mathcal{N}, \mathcal{D} \cup \mathcal{C}$)
10:    CREATEANDADDPROCESSGRAPH(process, mdt)
11: end while
```

Pay attention that we are not going to demonstrate the exaction of first three passes with this example. For this reason we will provide the BPMN generated with figure 3.8 after applying all 3 passes in figure 3.9.

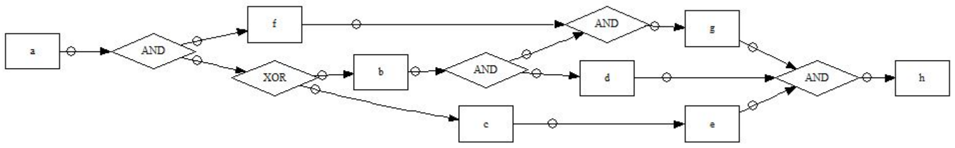


Figure 3.9: BPMN after third pass, result of the algorithm 5

Notice, that in the above 3.9 generated figure is the missing part of copy construct, which we are going to generate in this pass. Now, we need to proceed with the algorithm and reform the BPMN by adding all missing components.

First, we choose as *open* = {*g′*} workset of events which are copy and not cutoff. Correspondingly, the *curr* is selected from the *open* set, note that in the example currently we have only one copy but they can be multiple as well as some of the copies can have the same corresponding event. However, in this case, current selected element will be *curr* = *g′* and *g* corresponding of it. Note that, we will execute from the line 3 : 10 with one event each time until the *open* set will get empty.

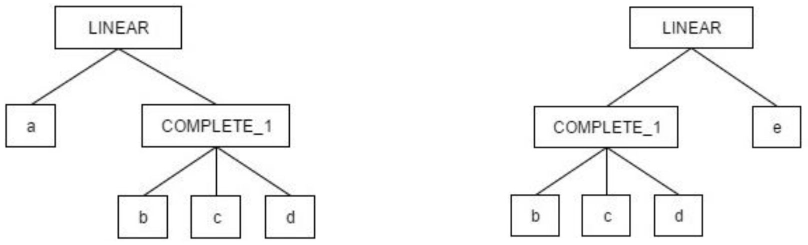
Next step in line 4, it is important to compute the P and Q sets carefully in this pass, as we do not want to include multiple duplicates that have the same corresponding into Q. For example, it can be the case that besides $p\prime$ we have other copy $p\prime\prime$ which is in conflict with other duplicates. In this case, if we consider $g\prime$ and $g\prime\prime$ being in the same Q set it will generate join and split getaway. By taking into account this fact, the Q will get the set of $\{g, d, c\}$ and accordingly P will be all direct predecessors of Q which is $\{f, b, c\}$.

The rest from the line 6 till 11, is replicating from the first pass and eventually generating the final BPMN.

3.2 Proceeding the MDTs

In this section, we will discuss how to proceed MDTs and do corresponding changes in BPMN model or add some components to it.

As we know from section 2.4 the modules in the MDT can be in 4 types $COMPLETE$ (C), $LINEAR$ (L), $TRIVIAL$ and $PRIMITIVE$(P).



(a) MDT with split gateway (b) MDT with join gateway

Figure 3.10: Example join and split MDTs

To proceed the computation of each leave in the tree we should start from the root of the tree and recursively go deeper until all nodes will be visited. To highlight, in the scope of this paper the root of the tree can be only $LINEAR$ or $PRIMITIVE$, as well as the leaves of the tree will be $TRIVIAL$.

To compute correctly all gateways and directions of arcs we need to carefully filter whether the module is in the left branch of the root or in the right branch of it. Because the MDT will be in the correct execution order always from left to right.

In general, our approach of generating BPMN from the MDTs are quite simple and straightforward.

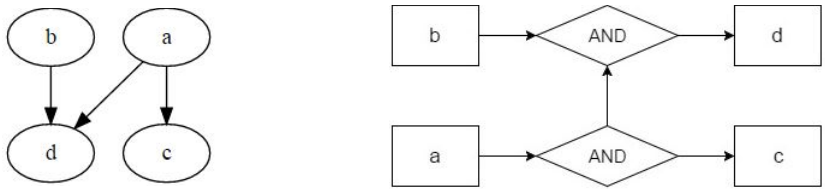
- **TRIVIAL :**
  T is the simplest cell (leaves) in the MDT, and in the words of BPMN it will be generated as a *Event* without any incoming or outgoing arcs. The connection of Ts will be done when we resolve rest of the modules.

- **COMPLETE :**
  As we know already C can be in two types, first when there the elements in C are connected, meaning they are in conflict and we call it $COMPLETE\_1$ (C_1). Second, the absents of the arcs between elements in C which means that they are concurrent and accordingly we call it $COMPLETE\_0$ (C_0).

  Consequently, in the case of the C_1 we will make XOR gateway and similarly in the case of C_0 we will make AND gateway. The decision whether it will be join or split depends in which order modules are in the MDT. For example, see figure 3.10a, we have C_1 in the right branch of the three, so accordingly it will add *XOR-split* gateway to BPMN. As follows, in the figure 3.10b we have the C_1 on the left-hand side, so it will add *XOR-join* gateway to the BPMN.

- **LINEAR :**
  L simply shows the linear connection between branches. However, in this work we will not have the case that L being in part of the module and we can find it only as a root module.

- **PRIMITIVE :**
  To resolve the primitive module can be a bit hassling as it is not that straightforward as the rest. However, let us go over this example of the simplest primitive module in figure 3.11.



(a) Primitive constructed BRG (b) BPMN form primitive

Figure 3.11: Example of primitive construct

In this case, we should add two AND gateways one after a which will be *split* point and another one before d, *join* point. In this way, we will have the primitive construct generated in BPMN. Note that in this example we have single Ts constructing primitive, in the real world more complex examples may come along. For instance, one case can be when instead of trivial parts of the primitive can be C_1 or C_2.

When all these changes will apply, we will have isomorphic BPMN of BRG. To point out, all the events will be mapped to original Transitions and by doing so we will never duplicate events.

3.3 Adding tau transition

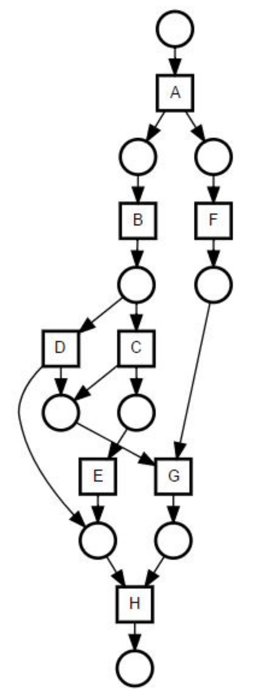
Exceptionally, there are cases when we should add tau transition to the unfolded net in order to have an isomorphic BPMN output.

Implicit place is a place which can be removed without changing the behavior of the net. The $P8$ in the figure 3.12b is clearly implicit place. In the literature [1], there are several techniques to remove implicit places from Petri net.

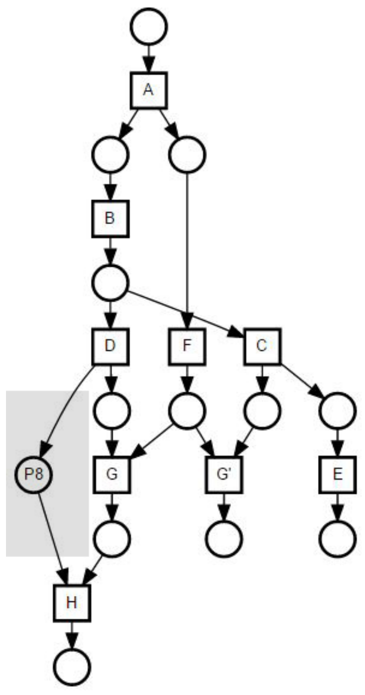
We assume that input Petri nets for our case do not contain implicit places. However, the unfolding of a Petri net may result in some *places* that are apparently implicit.

If we do not pay attention to this *places*, we would modify their behavior in the output BPMN.

In this cases, we should take into account the existences of the *implicit place* and add the corresponding event to BPMN. For example, the translation of the net in figure 3.12a will result the BPMN with *tau* transition.



(a) Input Petri net



(b) Unfolded net

Figure 3.12: Exception input Petri net with unfolding

Notice that, original net in the figure 3.12a does not contain the implicit place but after unfolding we can simply see that there is.

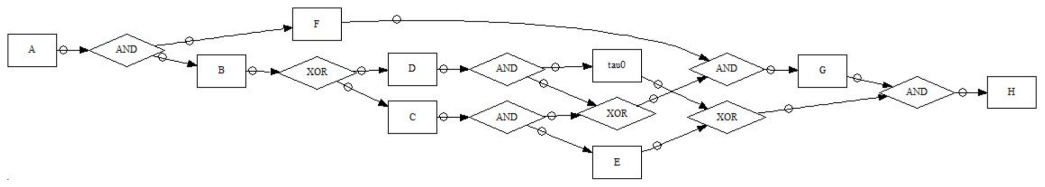


Figure 3.13: BPMN output with tau event

In the figure 3.13 below we can see the output Petri net, where it is noticeable the tau event was added in output BPMN because of implicit place was found in unfolding. Let us assume that we remove the tau event from the BPMN, it will obviously change the behavior of the process.

Chapter 4

Comparison and Results

To justify our approach we implemented the algorithm in the *The relationship between workflow graphs and free-choice workflow nets* [5]. In this section, we will run same tests on both our and their approach, to understand more clearly the performance and Petri net class coverage differences.

The translations and tests were performed on the laptop with, Dual cor 2.20 GHz processor and 16Gb memory. The used environment was Windows 8.1 64bit and Java virtual machine 1.8. In order to avoid load time in the results we did each sample 5 times, by removing first and the last result we took the average.

4.1 Dataset

The test collection has been selected from real life process models, which has been published for research purposes [4]. Essentially, we selected all sound models only from this collection, such that the result was ended up with 375 sound models.

By observing the dataset, we found that many methods are not workflow as there are cases when several input and/or output places exist. For this reason, we did changes in the models explicitly before executing them. First, we added sink transition and added all end places to it. Similarly, we added source transition and connected to all start places.

On the other hand, it was requirement form [3] that each label in the Petri net must be unique in order to be unfolded. Hence, to avoid duplicate labeling of nodes we decided to rename all nodes before proceeding.

4.2 Performance

To , we will run performance analyses with both algorithms. We decided to filter out all models which could be handled with both implementations. The result was 255 sound models with minimum 11 nodes/arcs, maximum 550 nodes/arcs and the average size 87 nodes/arcs complexity.

Then, each of this models was processed 5 times and we took the average execution time in milliseconds for each model by excluding the first and the last result.

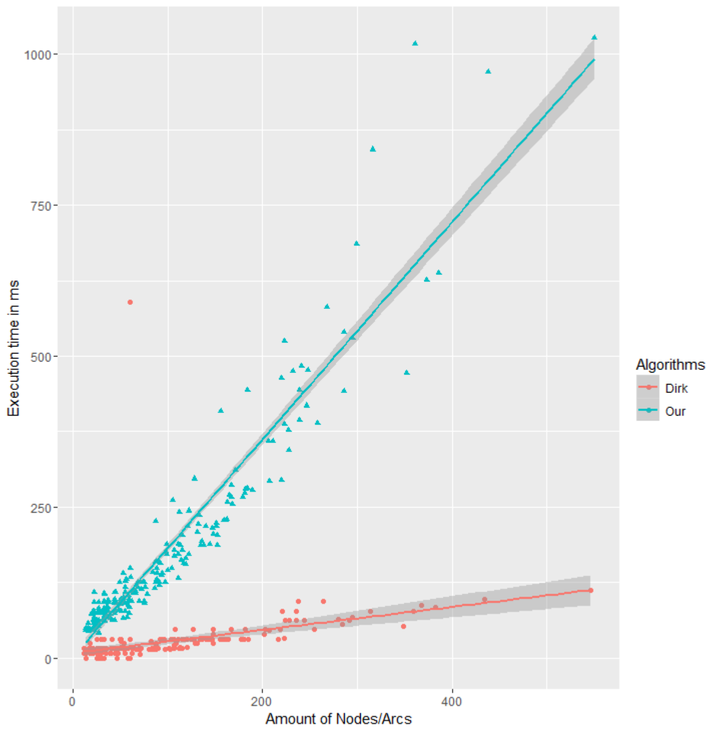


Figure 4.1: Performance plot of both algorithms

As we see in the figure 4.1 our approach execution time is exponentially growing with the complexity of the models. While in the first approach we see that it is linear and the complexity does not affect the execution time. More precisely, in our approach, the execution time depends on the complexity of the model, whereas in their approach the model complexity will not make much difference.

Nonetheless, the execution maximum time was around the second as an input 550 nodes/arcs model. Though in the second approach the same model was executed in around 100ms which is about 10 times faster than our approach. Since, we used the unfolding algorithm from [3] which is exponential, this is the main explanation of this huge differences in the performance. At the same time, our algorithm in every pass are linear, hence the performance would be similar to first approach.

Chapter 5

Conclusion

To conclude, the main interest of the paper was to study more the field of translating Petri nets to industrial languages (in our case BPMN) from the practical point of view. Surprisingly, we found out that there is only a small amount of research devoted to this problem. In the case of BPMN, the only exception we aware of is the [5], where they present bidirectional translation between free-choice workflow nets and the aforementioned notation. This contrast with fact that the other way translation is well understood.

In that context, the initial idea of the thesis aimed at implementing a translator based on the method described in [5]. However, after implementation, we found that the method did not cover more complex cases. Hence, we came to the idea of expanding the method to cover cases beyond the family of free-choice Petri net. This aim is justified by the fact that, in the general case, process discovery methods produce nets which are not necessarily free-choice. Thus, our contribution was to cover the case of Petri nets with symmetric confusion.

Our work concentrated in the practical side of the problem. Thus, we implemented our ideas in a prototype and run some preliminary performance experiments thereof. However, we did not tackled fully developed a theory to prove the correctness and completeness of the method. We consider that the theory is important but requires a deep understanding of several formalisms such as Petri nets, Net unfolding, Modular decomposition and, in general, concurrency theory. Therefore, such work is left open for further research. Finally, one natural extension would be to further study the case of Petri nets with asymmetric confusion.

Bibliography

[1] *Proceedings of the 8th International Workshop on Petri Nets and Performance Models, PNPM 1999, Zaragoza, Spain, September 8-10, 1999.* IEEE Computer Society, 1999.

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

[3] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of mcmillan's unfolding algorithm. In *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, TACAs '96, pages 87–106, London, UK, UK, 1996. Springer-Verlag.

[4] Dirk Fahland, Cédric Favre, Jana Koehler, Niels Lohmann, Hagen Völzer, and Karsten Wolf. Analysis on demand: Instantaneous soundness checking of industrial business process models. *Data Knowl. Eng.*, 70(5):448–466, 2011.

[5] Cédric Favre, Dirk Fahland, and Hagen Völzer. The relationship between workflow graphs and free-choice workflow nets. *Inf. Syst.*, 47:197–219, 2015.

[6] Stefan Haar. Clusters, confusion and unfoldings. *Fundam. Inform.*, 47(3-4):259–270, 2001.

[7] Kurt Jensen and Wil M. P. van der Aalst, editors. *Transactions on Petri Nets and Other Models of Concurrency II, Special Issue on Concurrency in Process-Aware Information Systems*, volume 5460 of *Lecture Notes in Computer Science*. Springer, 2009.

[8] Tadao Murata. Petri nets: Properties, analysis and appl kat ions. *IEEE*, 77(4):547 – 548, 1998.

[9] Object Management Group (OMG). *Business Process Model and Notation (BPMN) Version 2.0.*

[10] Object Management Group (OMG). *OMG Unified Modeling Language Specification.*

[11] Nathaniel Palmer. *XML Process Definition Language*, pages 3601–3601. Springer US, Boston, MA, 2009.

[12] Artem Polyvyanyy, Luciano García-Bañuelos, and Marlon Dumas. Structuring acyclic process models. *Information Systems*, 37(6):518 – 538, 2012.

[13] Wil M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):17 – 66, 1998.

[14] Wil M. P. van der Aalst. Workflow verification: Finding control-flow errors using petri-net-based techniques. In *Business Process Management, Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 161–183. Springer, 2000.

[15] Wil M.P. van der Aalst and Kristian Bisgaard Lassen. Translating unstructured workflow processes to readable bpel: Theory and implementation. *Information and Software Technology*, 50(3):131 – 159, 2008.

[16] Wikipedia. Modular decomposition — wikipedia, the free encyclopedia, 2017. [Online; accessed 29-April-2017].

Licence

Non-exclusive licence to reproduce thesis and make thesis public