Automated Discovery of Structured Business Process Models

Supervisors
Prof. Giorgio Bruno
Prof. Marcello La Rosa

Candidate
Adriano Augusto

ACADEMIC YEAR 2015-2016
Abstract

This thesis addresses the problem of discovering business process models from event logs. Existing solutions of this problem provide different trade-offs between accuracy and complexity of the discovered models. With respect to the latter criterion, previous empirical studies have shown that block-structured process models usually are simpler and consequentially are more understandable and less error-prone than unstructured ones. For this reason, several automated process discovery techniques generate block-structured models by construction. However, these techniques intertwine the concern of delivering accurate models with that of ensuring their structuredness, sometimes sacrificing the former to ensure the latter. In this thesis we present an alternative approach that separates these two concerns. Instead of directly discover a structured process model, firstly a well-known discovery technique is used to discover a more accurate but sometimes unstructured (and even unsound) process model, and secondly the resulting model is transformed into an equivalent structured one, improving in this way its understandability without sacrificing accuracy. Finally an experimental evaluation shows that this new “discover and structure” approach outperforms traditional “discover structured” approaches with respect to a range of accuracy and complexity measures.
Contents

List of Figures v
List of Tables vi
Statement of Original Authorship vii
Acknowledgements viii

1 Introduction 1

2 Background and Related Work 5
  2.1 Business Process Management .......................... 5
  2.1.1 BPM Life-Cycle ....................................... 6
  2.1.2 Business Processes ................................... 8
  2.2 Business Process Modelling Languages .................. 10
  2.2.1 Business Process Model and Notation .................. 11
  2.2.2 Apromore and Canonical Process Format ............... 17
  2.3 Process Mining ........................................ 19
  2.4 Accuracy and Complexity of Discovered Models .......... 25
  2.5 Process Structuring .................................... 27

3 Approach 30
  3.1 Discovery with BPMNMiner ............................... 30
  3.1.1 Mining ............................................... 31
  3.1.2 Cleaning ........................................... 31
  3.2 Structure with iBPStruct .............................. 32
  3.2.1 Structuring ......................................... 32
  3.2.2 Clone Detection ...................................... 39
  3.3 Complexity Analysis ................................... 42
4 Implementation and Deployment
  4.1 Apromore ......................................................... 46
  4.2 ProM ................................................................. 49
  4.3 iBPStruct .......................................................... 51
  4.4 Structured Miner .................................................. 54
  4.5 BPMNMiner 2.0 ................................................... 55
  4.6 Complexity Calculator for BPMN and CPF Models .............. 56

5 Evaluation and Results ............................................. 58
  5.1 Setup and Dataset ................................................ 58
  5.2 Results ............................................................. 61
    5.2.1 Heuristic Miner .............................................. 61
    5.2.2 Fodina Miner ............................................... 65
  5.3 Discussion ........................................................ 66
  5.4 Threats to Validity .............................................. 68

6 Conclusion .......................................................... 69

Bibliography ............................................................ 71
### List of Figures

1.1 Example of discovered model using Inductive Miner. .......... 2  
1.2 Example of discovered model using Heuristic Miner. .......... 3  
1.3 Example of discovered model using our approach on top of Heuristic Miner. ........................................ 4  
2.2 Amazon *Order-to-Cash* business process. ..................... 9  
2.3 Samples of Activities. ........................................... 11  
2.4 Collapsed Subprocess (above) and Expanded Subprocess (below). 12  
2.5 Samples of Events. .............................................. 14  
2.6 Example of Timer Boundary Event. .............................. 15  
2.7 An example of correct use of XOR-split and XOR-join (left), sample of deadlock (option to complete violation) (right). .. 16  
2.8 An example of correct use of AND-split and AND-join (left), sample of lack of synchronization (proper completion violation) (right). 17  
2.9 An example of correct use of OR-split and OR-join (left), an unsound process (right). ............................................. 17  
2.10 Process model and its event log. ................................ 21  
2.11 Event log sorted by Instance-ID. ............................... 22  
2.12 A process model discovered with Evolutionary Tree Miner. ... 24  
2.13 A process model discovered with Fodina. ...................... 24  
2.14 A process model decomposed in its top level SESE fragments. . 28  
3.1 Overview of the proposed approach. ............................ 30  
3.2 Examples of application of the three cleaning heuristics. .... 32  
3.3 Structuring of injection and ejection. ........................... 33  
3.4 An example application of the $A^*$ search tree with our structuring method. ................................................. 38  
3.5 A process model mined using Heuristic Miner. ................ 43
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6</td>
<td>Process model in figure 3.5 after the Structure</td>
<td>43</td>
</tr>
<tr>
<td>3.7</td>
<td>Process model in figure 3.6 after the first clone removal.</td>
<td>43</td>
</tr>
<tr>
<td>3.8</td>
<td>The structured process model in figure 3.6 after the clone removal.</td>
<td>44</td>
</tr>
<tr>
<td>4.1</td>
<td>Apromore architecture.</td>
<td>47</td>
</tr>
<tr>
<td>4.2</td>
<td>Apromore portal interface.</td>
<td>48</td>
</tr>
<tr>
<td>4.3</td>
<td>Apromore editor interface.</td>
<td>48</td>
</tr>
<tr>
<td>4.4</td>
<td>ProM data tab.</td>
<td>50</td>
</tr>
<tr>
<td>4.5</td>
<td>ProM execute tab.</td>
<td>50</td>
</tr>
<tr>
<td>4.6</td>
<td>iBPStruct interface.</td>
<td>53</td>
</tr>
<tr>
<td>4.7</td>
<td>iBPStruct plugin in ProM.</td>
<td>54</td>
</tr>
<tr>
<td>4.8</td>
<td>Complexity Calculator plugin in Apromore.</td>
<td>56</td>
</tr>
<tr>
<td>4.9</td>
<td>Complexity Calculator output.</td>
<td>57</td>
</tr>
<tr>
<td>5.1</td>
<td>Taxonomy of models discovered by Heuristic Miner from the logs.</td>
<td>60</td>
</tr>
<tr>
<td>5.2</td>
<td>Taxonomy of models discovered by Fodina Miner from the logs.</td>
<td>60</td>
</tr>
<tr>
<td>5.3</td>
<td>Discovered model from a SAP R/3 log using Inductive Miner, and</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>its accuracy metrics.</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Discovered model from a SAP R/3 log using Heuristic Miner, and</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>its accuracy metrics.</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>Discovered model from a SAP R/3 log using BPMNMiner 2.0 on top of Heuristic Miner, and its accuracy metrics.</td>
<td>65</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Accuracy of models discovered from real-life data with Heuristic Miner. ................................................. 62
5.2 Complexity of models discovered from real-life data with Heuristic Miner. ................................................. 62
5.3 Accuracy of models discovered from artificial data with Heuristic Miner. .................................................. 63
5.4 Complexity of models discovered from artificial data with Heuristic Miner. ................................................. 63
5.5 Accuracy before and after enabling the Pull-Up operator. ........ 63
5.6 Complexity before and after enabling the Pull-Up operator. .... 63
5.7 Accuracy of models discovered from real-life data with Fodina Miner. ..................................................... 66
5.8 Complexity of models discovered from real-life data with Fodina Miner. ..................................................... 66
5.9 Accuracy of models discovered from artificial data with Fodina Miner. ..................................................... 67
5.10 Complexity of models discovered from artificial data with Fodina Miner. ................................................... 67
Statement of Original Authorship

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Adriano Augusto
Acknowledgements

First of all, I want to thank the person without whom I would not be writing this acknowledgements for this thesis: my professor and supervisor at Polytechnic of Turin, Giorgio Bruno. I thank him for having helped me to accomplish my thesis project abroad at one of the most important institute of the academic world: the Queensland University of Technology (QUT).

Secondly, I would thank my supervisor at QUT, Marcello La Rosa, and professor Marlon Dumas at University of Tartu, for their guidance and their advices throughout the whole research project.

Also, a special acknowledgement is for Raffaele Conforti, for the great experience I had working with him in a research team and for the good moments we had in our spare time.

Lastly, but not last, I want to thank my parents Maria Concetta Busá and Antonino Augusto for their support, their help and their presence even being far away.
Chapter 1

Introduction

During our everyday lives, at almost any time, we are part of one or even more processes concurrently: sometimes as resources, sometimes as customers, sometimes as simple data. Usually we do not notice it, for we do not realize we are acting within a process or for we are just not interested in insights of the process we are temporarily involved.

On the other hand, companies and industries all over the world are getting more interested on studying and understanding the inner processes on which they rely for their business. Indeed, on these processes depend the quality of their services, their incomes, their costs, the satisfaction of their clients and consequentialy their success. In this environment, Business Process Management (BPM) is born and it stands as lighthouse to help these industries analysing and assessing their business processes and possibly improving them.

First of all, as us, companies sometimes do not have a proper knowledge of their internal process architecture and its life-cycles. Despite managers and employees are aware they are part of processes and can even have a dim idea about their role and how tasks succeed each other, most of the times they do not have a global view of the whole process and/or how different processes collaborate and interact. In order to find out more about their business processes, industries began using automated techniques to extract models of their processes observing, recording and monitoring (through event logs) their normal execution.

In this context jumps in the Process Mining, a modern branch in information technology field, whose main target is to research, develop and provide methods, algorithms and techniques able to analyse a given event log and pull out from it a business process model [vdA11b]. The simplest event log is a set of traces, each consisting of a sequence of events observed within one execution of a process.
Figure 1.1: Example of discovered model using Inductive Miner.

However, event logs can store every kind of data linked to each recorded event, such as: starting time, ending time, resources and data. The more is the info stored, the deeper can go the analysis of the process from which the event log has been generated. Therefore, a more accurate model could be produced as output of this mining procedure.

Over the last years many automated process discovery methods have been developed to tackle this problem, but so far none of them can guarantee to deliver the perfect process model as result, so that they usually strike various trade-offs between accuracy and complexity of the output model [WBVB12]. In this scenario, accuracy consist of three dimensions:

i. **fitness**: how many traces, of the input log, the discovered model is able to replay;

ii. **precision**: how many traces, generated on top of the model, match with the traces in the input log;

iii. **generalization**: to what extent is the model able to parse traces that, despite not being present in the input log, can actually be produced by the process under observation.

Whilst complexity is commonly measured via size metrics (e.g. number of nodes composing the model) and structural metrics. The latter quantify either the amount of branching in a process model or its degree of structuredness (the extent to which a model is composed of well-structured single-entry, single-exit components), which have been empirically shown to be proxies for understandability of a process model [Men08].

Inspired by the observation that structured process models may be more understandable than unstructured ones [DLM+12], several automated process discovery methods deliver structured models by construction [LFvdA13, Bvv12]
MRG15], but doing so, the discovered model could turn into an approximation of the original one whether the real process itself was not structured.

Approaches like these intertwine the concern of accuracy with that of structuredness, sometimes sacrificing the former to achieve the latter and vice-versa. For instance, figure 1.1 shows a process model discovered from an event log by one of these methods, specifically Inductive Miner [LFvdA13]. Despite the output model has a fitness and generalization of 1.00 on 1.00, it lacks in precision with a score of 0.22 on 1.00. This happened due to the fact Inductive Miner tried to get a block-structured model whilst the process hidden in the input event log was not, ending up with a model that is way too general. Alternatively, figure 1.2 displays a result obtained from the same event log but using Heuristic Miner[WR11] as discovery algorithm. Since this latter does not set any constraint on the structure and complexity of the output process model, even generating a more tangled one, it outperforms Inductive Miner with a score of 1.00 for all the accuracy metrics.

Considering the just presented background, the problem that this thesis aims to address is the following: obviate this trade-off between accuracy and complexity. As solution an automated process discovery approach that generates structured models, yet achieving essentially the same (or even better) fitness, precision and generalization has been designed and implemented. This new method [ACD16], follows a two-phased approach:

i. firstly, a model is discovered from the input log using a process discovery algorithm that consistently delivers accurate but potentially unstructured and even unsound models, for example Heuristic Miner;

ii. secondly, the discovered model is transformed into a sound and structured model combining three techniques: a technique to maximally block-structure acyclic process models, an extended version of a technique for block-structuring flowchart and a technique of clone detection and removal.
Figure 1.3: Example of discovered model using our approach on top of Heuristic Miner.

Figure 1.3 shows, for sake of completeness, the process model got from the same log of the previous two examples, but applying our method using Heuristic Miner during the first phase. Basically the model is the one in figure 1.2 automatically transformed into its equivalent structured through the second phase of our approach.

Finally, the presented approach has been evaluated extensively on real-life and synthetic event logs, consolidating its performance relatively to two representative methods that discover structured models by construction: Inductive Miner and Evolutionary Tree Miner [MRG+15].

The rest of this thesis is organized as follows. Chapter 2 introduces the context of our problem: Business Process Management and Process Mining, in particular focusing on business process model representations, existing automated process discovery algorithms, techniques to evaluate the quality of the discovered models in terms of accuracy and complexity, and methods for structuring process models. Chapter 3 introduce our proposed approach and how it operates. Chapter 4 gives an overview on how it has been implemented and in which frameworks can be found and used. Lastly chapter 5 reports on the empirical evaluation and chapter 6 summarizes the contributions and outlines future work directions.
Chapter 2

Background and Related Work

In this section we firstly give a general overview on Business Process Management and business process model representations. Then we present Process Mining and some of the most common automated process discovery algorithms, classifying them into two main categories: those which discover structured model by construction, those which do not. Also we explain how is possible evaluate the quality of the discovered models through a set of accuracy and complexity metrics. Finally, we introduce two complementary methods for transforming unstructured process models into structured ones, which we will later use as building blocks for our proposal.

2.1 Business Process Management

Business Process Management (BPM) is the science whose aim is to analyse, monitor and possibly improve business processes within an organization, regardless its type, its provided services, characteristics and targets [DLRMR13]. For its aim, this science orbits around two distinct main fields, and many different contexts, that at a glance could not appear linked in anyway.

The first main field BPM is bound to, as its name says, is: business. It consequentially means dealing with resources and data of a company and in general with business stakeholders. The reason why BPM attracts the latter is for its proven capabilities to help improving (in various ways) the “results” of their business. Dependently on the company type and its objectives the “results” it would obtain and improve may widely vary along with the context. For instance, an organization trading on-line may be interested in increasing its income and reducing its costs, or short supplying and shipping times. A financial company who lends money, would be interested in finding patterns of the behaviours and/or
profiles of their customers in order to prevent or detect in time possible breach on ongoing loans. Similarly, insurance companies would definitely like to somehow forecast whether a client eventually leads to a lost or a gain of money studying the instances of their former and still clients.

However, BPM is not a forecast science and, in order to obtain satisfactory benefits, it relies on applying analysis and monitoring methods to business processes and their data. This often entails managing huge amounts of information, so that automated technologies, being more precise and faster, are preferred over manual ones. Here follows the second main field BPM is bound to: information technology (IT). Since sciences like data mining and artificial intelligence have for years been working on many analysis methods and algorithms to handle and exploit big data, IT specialists from these fields perfectly fit the BPM scenario and play the most important role on its stage, leading to a synergy with business stakeholders in the context of a BPM project.

2.1.1 BPM Life-Cycle

A BPM project usually begins with a simple happening: business stakeholders of an organization realize something seems not working properly or could work better in its business, so that suddenly they ask IT specialists of BPM for advices. What follows next is showed in figure 2.1, which describes the BPM life-cycle [DLRMR13]. Simply, this diagram formally explains how business stakeholders and IT specialists collaborate together through distinct phases in order to achieve a possible business improvement.

- **process identification**: first of all, a business process (or a set of processes) has to be addressed as subject of improvement. This has to be the process related to the suspected issue and/or weakness identified by the business stakeholders. Despite the identification of this process may seem easy, it is not always trivial, because processes in an organization are tangled together and figure out the boundaries of each of them can be tricky, especially when the business stakeholders have not a deep knowledge of their internal processes. The output of this phase is the so-called *process architecture*: a set of processes and their relationships, that outline how work is performed within the organization.

- **process discovery**: finally once got the process architecture, a process (or a subset of processes) can be chosen accordingly to their relevance with
the suspected issues or weaknesses (such as bottlenecks, risky conditions, deadlocks, etc.), that the organization wants to fix or improve. When the process is selected, it has to be discovered as a model: a graphical view of the business process, which easily represents its workflow. This representation of the process is named: *as-is process model*.

- **process analysis**: the IT specialists study and inspect throughout the process model using a set of automated, and sometimes even manual, analysis techniques. As result, the suspected issues and/or weaknesses are detected and whenever possible assessed through an evaluation of performance metrics (e.g. execution time of a task or of the entire process). Final output of this analysis is a list of issues and weaknesses, usually sorted by the relevance of their impact on the overall performance.

- **process redesign**: in this phase the IT specialists take into consideration possible changes to apply to the *as-is process model* in order to resolve the issues and weaknesses found during the previous phase. Where many options are available to fix one or more issues, they are compared each other through a further analysis in order to finally pick the best solution which could better meet the objectives of the business stakeholders. Once all the issues (or most of them) are solved, the taken solutions have to be applied to the as-is process, so that the process is redesigned. At the end of this redesigning procedure a new model will take over the as-is process model, named: *to-be process model*.

- **process implementation**: at this stage, all the fixes and improvements are done on the model, but they have still to be implemented in order to assess if the results expected hold. Therefore, the *as-is process model* is turned into the *to-be process model* applying all the designed changes within the organization workflow. Usually this entails: changing how the process was performed (e.g. introducing new tasks, swapping resources, splitting or merging previous activities, adding parallel workflow etc.); and deploying or upgrading the IT systems needed to monitor the new process now on and give support to the next and last phase of the BPM cycle.

- **process monitoring and controlling**: even though the new process has been implemented following the *to-be process model* guidelines, something could diverge from the expectations and mismatch the theoretical performances. For this reason, the process is automatically monitored and its
2.1. BUSINESS PROCESS MANAGEMENT

Figure 2.1: Business Process Management life-cycle [DLRMR13]

Data is continuously collected using the IT systems. Through this monitoring, IT specialists carry out on-line and off-line performance analysis of the new process, which goal is: evaluate if the improvements are effective and if the weaknesses have been reduced or removed.

Nevertheless the implementation of the to-be process could give the results expected, new deviations or errors may show up soon or later, consequently the cycle would begin again.

Presenting further and deeper each of the BPM life-cycle phases is out of the scope of this thesis. We will focus only on what is related to our topic and useful for its understanding, that is the discovery phase. However, before stepping into this latter, we need to introduce what exactly is a business process and how it can be modelled.

2.1.2 Business Processes

“Business processes are what companies do whenever they deliver a service or a product to customers.” [DLRMR13]

In order to better understand what means this sentence, let us consider an example of a moderate accurate view of a specific business process of the popular
2.1. BUSINESS PROCESS MANAGEMENT

Figure 2.2: Amazon *Order-to-Cash* business process.

e-commerce company Amazon (represented in figure 2.2 using Business Process Model and Notation as process modelling language).

- **Order-to-Cash**: let us suppose an Amazon user is filling his digital cart with a certain number of items he wants to purchase, as soon as he gets everything he needs, he will checkout. At the click of the button “checkout” this business process is triggered, so that we can identify “checkout requested” as start event of this business process. What follows next is a sequence of well defined activities: the user is asked for the shipping address, for the shipment type (standard, express, etc.) and for the payment method and details. Then, Amazon checks the validity of the payment details and if they are valid it sends a confirmation email, otherwise it denies the order and the process would end at this point, generating an “order aborted” ending event. After the confirmation email, the items are prepared for the shipping and once ready the payment is charged. If the payment is charged correctly the items are shipped and a receipt is sent to the user via email, so that “order fulfilled” will be the second and correct end event of this business process, otherwise if the payment fails the order is aborted.

This simple example is a business process which focus on the activities between a checkout request, related to a client’s order, to the fulfilment of the order. The name: *Order-to-Cash* summarizes the input and the output of this process, which are respectively: the client’s order and the money Amazon gains.

However, we can also describe a business process less accurately, more generically, as for the next examples:

- **Issue-to-Resolution**: this business process is common in all those organizations providing a warranty or customer service. It starts when a client requests assistance because he is experiencing an issue with a previously purchased item or requested service, and hopefully it ends with the resolution of the reported issue.
- **Application-to-Approval**: everyone has been part of this process at least once in his life, indeed this is the classic scenario of someone applying for something. It may be an application for an educational institution, rather than an application for a home loan. It may end with the approval of the submitted application or, in certain cases, with a rejection of it when some mandatory criteria for the grant are not met while processing the application.

- **Procure-to-Pay**: another business process present in all trading companies. It is triggered when a product is running out and an order has to be submitted to the supplier. It ends when the items are paid and again available in stock for selling.

In conclusion, regardless the accuracy and the modelling language used to describe a business process, we can say it is: a sequence of activities triggered by a specific event bound to a certain input, that leads to another specific event bound to an output. Despite in all the just presented instances of business processes it is easy identify input and output, it is not realize what exactly happens in between. Indeed, depending on the company, the sequence of activities which drive from the start event to the end event may drastically vary, and along with them obviously vary the quality of the service and the performance of the process. Also this latter explains why BPM has become so popular, and why certain companies succeed in their business better than others, nevertheless they are of the same type and are built on the same (generic) business processes.

### 2.2 Business Process Modelling Languages

Analysing a business process would be almost impossible without a proper graphical representation, and even with that it sometimes could be not trivial at all if the process would exceed certain complexity metrics [Men08]. This means that for a good understanding of a business process we need to model it and we need to model it as simply as possible.

Nowadays, many process modelling languages are available for this purpose, such as: Petri Nets (PNs), Event-Driven Process Chains (EPCs), Yet Another Workflow Language (YAWL), Business Process Model and Notation (BPMN), Canonical Process Format (CPF). Despite each of these modelling languages can be used to model a process, depending on the process characteristics and on which insights we are interested most, some of them would make easier than
others the modelling. Indeed, these languages basically differ each other on how they model process characteristics, its control flows, and how complex would be the modelling phase and the final model.

To what concerns this thesis we will focus only on BPMN and CPF, starting giving a wide overview of the former and then a quick glance at the latter.

### 2.2.1 Business Process Model and Notation

BPMN is based on classic graphical workflow charts. It can easily model almost all kind of business process behaviours, control flows, data and resources relationships and interactions.

Here we present some of the most important elements of BPMN, leaving further details (of no use for the understanding of this thesis) to the BPMN 2.0 specification [Obj11].

- **Activities:**

  they allow to map tasks executed within the process by a resource, this latter can be a human actor or even an automatic system. To model activities we use labelled boxes. Their labels describe the task using the labelling rule: imperative verb + noun (e.g. send invoice). Additional markers can give further information about the type of activity. We distinguish the following types of activity (see figure 2.3, respectively left to right):

  - **Task:** this is the simplest activity, it refers to the execution of a single task by a resource.

  - **Parallel Multi-Instance:** this models an activity that can be executed concurrently as many times as the number of input data. Each instance of the activities in parallel is executed by the same type of resource but different actors. Figure 2.3 shows the example of an assessment.
2.2. BUSINESS PROCESS MODELLING LANGUAGES

Figure 2.4: Collapsed Subprocess (above) and Expanded Subprocess (below).

- Sequential Multi-Instance: this represents an activity that can be executed sequentially as many times as the number of input data. Each instance of the activities of the sequence is executed by the same actor. In figure 2.3 is displayed the example of a professor who has to mark the assignments of his students, he will mark an assignment after the other until all are marked.

- Simple Loop: this activity can be executed an unlimited number of times before the workflow of the process continues with the next activity. A condition that makes the loop ending has to be provided, in the example in figure 2.3 would be the choice of an username still available.

- Collapsed Subprocess: this is a place holder for another standalone process within the current process. It means this is not a single activity but a process itself, which usually is collapsed for readability reasons or for having no knowledge about what exactly happens within the subprocess, despite it is known it is not a single task.

- Expanded Subprocess: it is the result of the expansion of a collapsed subprocess, displaying all the activities within it. Figure 2.4 shows two views of the same process: one with a collapsed subprocess and the second with the expanded subprocess.
2.2. BUSINESS PROCESS MODELLING LANGUAGES

- **Events**: they allow to map happenings occurring during a process execution. Their shape is always a circle, but the border of the circle and the inner symbol change depending on type and trigger of the event (see figure 2.5).

Firstly, we distinguish two event usages:

- *Catching Event*: these events represent an happening that is caught from outside the process execution (e.g. message received);
- *Throwing Event*: these events represent an happening that is thrown outside the process execution (e.g. message sent).

Further, we have three event types:

- *Start Event*: is an event that triggers the process execution, its usage is always *catching*;
- *Intermediate Event*: is an event that happens in between the process execution, its usage can be either *catching* or *throwing*;
- *End Event*: is an event that ends the process execution, its usage is always *throwing*.

Finally, each of these events have a specific trigger that exactly describes the happening, the most common triggers are:

- *None*: it is used to map a non specific happening, it is often used for start or end events; it has no marker;
- *Message*: it is used to represent the receiving or the sending of a message, respectively the event would be a *catching* or a *throwing* event; its marker is an envelop;
- *Timer*: it is used to specify an exact date and/or time, so that once it is reached the event fires, this trigger can be used only for *catching* events; its marker is a watch;
- *Error*: it is used to model an error occurred while executing a process, it is used only as *catching* event, mostly as boundary event (later discussed); its marker is a thunderbolt;
- *Terminate*: it is used only for end events, it maps the need to immediately end all the activities in the process execution; its marker is a black bold circle.
A special case of intermediate events are the *boundary events*. These events have to be attached to subprocesses (either collapsed or expanded) and their usage is always catching. As soon as a boundary event is caught within the execution of the subprocess to which is attached, this latter instantaneously terminates and the execution continues from the boundary event. An example of boundary event is shown in figure 2.6. In this case it is a *timer boundary event* set at the 1st of August, supposing this the deadline to ship the application envelope. The meaning is the following: if while preparing the application envelope the 1st of August is reached, this subprocess terminates and the flow continues from the timer event, that leads directly to the termination of all the activities of the process (*terminate event*).

- **Sequence Flows**: they describe the order in which activities and events will be executed. Their shape is an arrow, and they can be labelled in order to highlight data conditions in case of exclusive branches (see figure 2.2).

- **Gateways**: they are control flow elements and are used to model exclusive, concurrent and inclusive behaviours. Their shape is a diamond regardless the captured behaviour, but also in this case depending on the mapped behaviour their inner symbol changes.

    - **XOR Gateways**: these gateways model an exclusive behaviour, they...
2.2. BUSINESS PROCESS MODELLING LANGUAGES

Figure 2.6: Example of Timer Boundary Event.

are marked with an \( \times \). A XOR gateway can either be a XOR-Split gateway (when the number of outgoing flows is greater than one) or a XOR-Join gateway (when the number of incoming flows is greater than one). The former represent a decision, so that only one outgoing flow can be taken to proceed with the process execution, usually the choice is made upon data conditions reported as labels of the outgoing flows. The latter is used to merge incoming flows, and it lets the execution continue as soon as one incoming flow has completed.

- **AND Gateways**: these gateways model a concurrent behaviour, they are marked with a \( + \). An AND gateway can either be an AND-Split gateway (when the number of outgoing flows is greater than one) or an AND-Join gateway (when the number of incoming flows is greater than one). The former let a parallel execution start, so that the process proceeds on all the outgoing flows concurrently. The latter is used to synchronize parallel incoming flows, and it lets the execution continue only when all the incoming flows have completed and ready to proceed.

- **OR Gateways**: these gateways model an inclusive behaviour, they are marked with a \( \bigcirc \). An OR gateway can either be an OR-Split gateway (when the number of outgoing flows is greater than one) or an OR-Join gateway (when the number of incoming flows is greater than one). The former represent a decision, but differently than the XOR-split gateway it allows the process execution to continue on one or on more (concurrently, up to all) the outgoing flows, however, also in this case this selection is usually done upon data conditions. The latter is used to merge incoming flows, and it lets the execution continue only when all the incoming executing flows have completed. Where, for “all
incoming executing flows\footnote{ incoming executing flows” we refer all the flows that were triggered by an OR-Split (or a combination of XOR-Split and AND-Split gateways).} further, gateways are responsible for the soundness of the process model. indeed, when used correctly the gateways guarantee the soundness of the model, otherwise the process can be unsound. a process model is sound if the following three criteria hold altogether:

- **option to complete**: each possible process instance must be able to reach the end event;
- **proper completion**: each end event cannot be triggered more than once in a process instance;
- **no dead activities**: for each activity there must be a process instance where such activity is executed.

if one of them is violated, the process is unsound.

figure 2.7 shows on the left a sound process model, where its possible executions are the followings: \{start, A, end\}, \{start, B, end\}. instead, the process on the right is unsound due to a deadlock which violates the option to complete criterion. indeed, the xor-split matches an and-join, which will be waiting forever a synchronization between A and B, that are exclusive activities.

another example is displayed in figure 2.8, also in this case on the left we have a correct use of and gateways within a process model that allows the behaviours: \{start, A, B, end\}, \{start, B, A, end\}. whilst the process on the right is unsound because it violates the second criterion. indeed, when the end event is reached, still A or B will be executing because of the lack of synchronization.
2.2. BUSINESS PROCESS MODELLING LANGUAGES

Figure 2.8: An example of correct use of AND-split and AND-join (left), sample of lack of synchronization (proper completion violation) (right).

Figure 2.9: An example of correct use of OR-split and OR-join (left), an unsound process (right).

Last example (figure 2.9) takes in consideration the OR gateways. On the left, we have a sound process that allows the following workflows: \{start, A, B, end\}, \{start, B, A, end\}, \{start, A, end\}, \{start, B, end\}. On the right, if the join gateway is an AND-join or a XOR-join, the process would be unsound. In the former case we would have a deadlock, in the latter we would have lack of synchronization. ¹

The remaining BPMN elements such as: Pools, Swimlanes, Data Objects, Annotations are used to model respectively: organization, resources, data and notes. However, these elements (along with some extensions of the previous presented elements) will not be taken into account in this thesis.

2.2.2 Apromore and Canonical Process Format

Since in the last decades business process models became more popular, many companies such as banks, insurances, hospitals and municipalities demanded

¹Despite it is not often used, the OR-join thanks to its semantic can correctly replace either a XOR-join or an AND-join.
for repositories to safely store their models. Consequentially, a proper storage technology needed to be developed and implemented, firstly to let data be securely and quickly accessible, secondly to apply strategies to reduce the memory usage. The results of this need are process model repositories like Apromore (Advanced Process Models Repository) [LRRVDA+11]: an open and extensible platform where any type of process models (BPMN, EPCs, YAWL, PNs) can be stored and possibly disclosed. Further, Apromore embeds a process model editor for the above listed modelling languages and allows to deploy OSGI plugins for process analysis. Finally, Apromore also concerns about the memory saving problem, that is struck using clone detection techniques [UDGBLR11]. Indeed, since process models often share clone fragments, it is enough save models as fragments allowing only one copy for each clone of them. This avoids data overlapping and ensure memory optimization. Successively, a process model is rebuilt at access time.

In Apromore environment was born and still stands the Canonical Process Format (CPF) [LRRVDA+11]. In spite we have previously mentioned CPF along with the other modelling languages, we have to say that it is not exactly a modelling language, since it does not own any graphical elements. We can consider it more likely a map, indeed, it was firstly introduced as a meta-model which allows to parse each model (independently of its native modelling language) into a standard form represented through CPF, where the process graphical elements still remains those of its native representation (attached to the CPF model as annotations). As consequence, since each process modelling language can be mapped on CPF, each process model can be turned into any other representation, with a minimum loss of information due to cross-language reasons.

As example, let us suppose we have an EPC model that we want to turn into a YAWL model, we can do it through CPF, simply parsing EPC into CPF and then CPF to YAWL. Due to the fact CPF contains all the core structural features shared by all the modelling languages, conversions like this are simple, despite could entail a loss of information for those non-shared features between (in this example) EPC and YAWL.

Further, CPF allows to refer to each process model independently of its modelling language through standard methods built on its meta-model. Indeed, thanks to the CPF implementation as Java object (within Apromore), it provides a fast and easy access through standard APIs to every model and all its elements. This because, each time any model is uploaded into Apromore, it will be firstly
converted into CPF and secondly stored in both its representation: the native one and the CPF one. Where the latter can be easily exploited as input of analysis OSGI plugins that can run into Apromore, and the former is referred when the model has to be displayed in its native graphical elements.

Apromore and CPF have been used as part of the implementation and the evaluation of the proposed approach presented in this thesis.

2.3 Process Mining

At this point, we have all the elements to go deeper into the discovery phase of the BPM life-cycle. As already said, in this phase one or more as-is process models have to be discovered. To achieve this goal there are two options: the first is through meetings and brain storming with business stakeholders and employees involved in the processes as resources, the second is automatically through the help of business process discovery techniques. This latter option is also known as Process Mining [VDA11a]. As the reader can easily guess, this is also the most preferred, but in order to make for it, data is needed, to be precise: event logs.

Everyday many business companies and public organizations generate an incredible quantity of data during the executions of their internal processes. This data is often stored inside one or more event logs. Naively, we can think at an event log as a file containing information about what happened and how within an organization (e.g. activities performed by employees, their chronological order, their execution times, their associated data), so that an event log actually contains the history of processes executions. Commonly a company tracks each process independently, therefore each process will have its own log. Nevertheless, if it happens the company has only a unique event log for all the processes, it would still be possible extract sub event logs in order to focus only on one process.

Formally, an event log is a set of process traces, a process trace is a chronological sequence of events observed in a process instance, and an event is a record that something occurred.

An event must have at least three attributes: the event type (i.e. the label/name of the event), the time it has been recorded, an identifier of the process instance within it happened. Also an event can report about the resource that generated it and the data it was linked. The more is the info stored, the deeper
can go the discovery algorithm and the more accurate will be the output process model. Figure 2.10 displays a synthetic process model and its event log. However, in its primitive form a log is unreadable. In order to understand its content, it is necessary at least sort it by the Instance-ID attribute as shown in figure 2.11. Then, each trace of the process can be extracted obtaining the log as a simple set of traces: \( \text{Log} = \{ \{\text{start, A, B, D, E, F, end}\}, \{\text{start, A, B, D, F, E, end}\}, \{\text{start, A, C, D, E, F, end}\}, \{\text{start, A, C, D, F, E, end}\} \} \). Further, this log is also a complete log. An event log is said complete when it contains all the possible traces the process can generate. If an event log is complete, it means all the possible behaviours of the process have been recorded. For complete logs the process discovery is usually easier. Finally, it is worth mention that in real-life event logs it is common to find noise, noise in an event log is the presence of recorded events that in the process model could not happen at the recorded time or at all. Considering the process model in figure 2.10, the trace: \( \{\text{start, A, B, C, D, E, F, end}\} \) contains noise because \( B \) and \( C \) cannot appear in the same process instance. Of course, there is no need to say that noise makes process discovery more difficult.

Despite process discovery is the core of process mining, it does not concern only about that. Also it provides analysis methods to evaluate the quality of the discovered process models in terms of accuracy and complexity. Before introducing these latter, we give an overview of the most common process mining algorithms, differentiating them in two types: those which discover structured models by construction, and those which do not.

A model is considered structured if it is made up only of SESE (single entry single exit) regions, possibly nested. Figure 2.12 shows a fully structured model highlighting all the non trivial SESE regions, while figure 2.13 an unstructured one, both discovered from the same real-life event log\(^2\), but respectively with: \textit{Evolutionary Tree Miner} and \textit{Fodina Miner}. As later discussed, to obtain a structured model, Evolutionary Tree Miner had to add some clones of activities.

The following are process mining algorithms which do not produce structured model by construction:

- \( \alpha \)-\textbf{Miner} [vdAWM04]: it is one of the earliest process mining algorithm. It is based on the use of the \textit{direct follow dependency}, that is built using the condition \( a > b \), which holds if an event of activity \( a \) directly precedes an

\[^2\text{For sake of space and readability the original activity labels have been replaced.}\]
2.3. PROCESS MINING

Figure 2.10: Process model and its event log.

<table>
<thead>
<tr>
<th>Instance-ID</th>
<th>Event Type</th>
<th>Time-stamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>start</td>
<td>16:00:07</td>
</tr>
<tr>
<td>1</td>
<td>start</td>
<td>16:01:19</td>
</tr>
<tr>
<td>0</td>
<td>A</td>
<td>16:02:42</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>16:04:23</td>
</tr>
<tr>
<td>0</td>
<td>B</td>
<td>16:05:36</td>
</tr>
<tr>
<td>2</td>
<td>start</td>
<td>16:10:09</td>
</tr>
<tr>
<td>0</td>
<td>D</td>
<td>16:11:08</td>
</tr>
<tr>
<td>3</td>
<td>start</td>
<td>16:13:20</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>16:15:11</td>
</tr>
<tr>
<td>0</td>
<td>E</td>
<td>16:17:34</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>16:19:22</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>16:25:41</td>
</tr>
<tr>
<td>0</td>
<td>F</td>
<td>16:30:25</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>16:31:12</td>
</tr>
<tr>
<td>0</td>
<td>end</td>
<td>16:45:16</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
<td>16:45:43</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>16:48:02</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>16:49:15</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>16:50:17</td>
</tr>
<tr>
<td>1</td>
<td>E</td>
<td>16:51:05</td>
</tr>
<tr>
<td>1</td>
<td>end</td>
<td>16:52:44</td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td>17:01:14</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>17:06:25</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>17:07:52</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>17:10:01</td>
</tr>
<tr>
<td>2</td>
<td>end</td>
<td>17:11:11</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td>17:15:36</td>
</tr>
<tr>
<td>3</td>
<td>end</td>
<td>17:17:42</td>
</tr>
</tbody>
</table>
2.3. PROCESS MINING

<table>
<thead>
<tr>
<th>Instance-ID</th>
<th>Event Type</th>
<th>Time-stamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>start</td>
<td>16:00:07</td>
</tr>
<tr>
<td>0</td>
<td>A</td>
<td>16:02:42</td>
</tr>
<tr>
<td>0</td>
<td>B</td>
<td>16:05:36</td>
</tr>
<tr>
<td>0</td>
<td>D</td>
<td>16:11:08</td>
</tr>
<tr>
<td>0</td>
<td>E</td>
<td>16:17:34</td>
</tr>
<tr>
<td>0</td>
<td>F</td>
<td>16:30:25</td>
</tr>
<tr>
<td>0</td>
<td>end</td>
<td>16:45:16</td>
</tr>
<tr>
<td>1</td>
<td>start</td>
<td>16:01:19</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>16:04:23</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>16:19:22</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
<td>16:45:43</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>16:49:15</td>
</tr>
<tr>
<td>1</td>
<td>E</td>
<td>16:51:05</td>
</tr>
<tr>
<td>1</td>
<td>end</td>
<td>16:52:44</td>
</tr>
<tr>
<td>2</td>
<td>start</td>
<td>16:10:09</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>16:15:11</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>16:48:02</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>16:50:17</td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td>17:01:14</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>17:10:01</td>
</tr>
<tr>
<td>2</td>
<td>end</td>
<td>17:11:11</td>
</tr>
<tr>
<td>3</td>
<td>start</td>
<td>16:13:20</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>16:25:41</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>16:31:12</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>17:06:25</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>17:07:52</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td>17:15:36</td>
</tr>
<tr>
<td>3</td>
<td>end</td>
<td>17:17:42</td>
</tr>
</tbody>
</table>

Figure 2.11: Event log sorted by Instance-ID.
event of activity \( b \) in at least one trace of the event log. Using \( a > b \) the following three ordering relations are defined:

i. *causality*, represented as \( a \rightarrow b \), is detected if \( a > b \) and \( b \not> a \);

ii. *conflict*, represented as \( a \# b \), is detected if \( a \not> b \) and \( b \not> a \);

iii. *concurrency*, represented as \( a \parallel b \), is detected if \( a > b \) and \( b > a \).

These ordering relations are then used to discover the process model.

- **Heuristic Miner** [WR11]: this algorithm was proposed in order to address some limitations affecting the \( \alpha \)-miner algorithm. Indeed, in order to deliver good results the \( \alpha \)-miner algorithm requires a complete event log free of noise. To overcome this limitation a frequency-based metric, i.e. \( a \Rightarrow b \), is introduced. The metric, defined as \( a \Rightarrow b = \left( \frac{|a > b| - |b > a|}{|a > b| + |b > a| + 1} \right) \), is used to verify if a \( \parallel \) relations is correctly identified. Whenever, the \( \parallel \) relation is identified, it is replaced by the \( \rightarrow \) if the value of \( \Rightarrow \) exceeds a given threshold. Unfortunately, despite this countermeasure, its output may be unstructured and even unsound.

- **Fodina Miner**\(^3\) [Bvv12]: this algorithm as Heuristic miner is based on heuristics, it has been developed as extension/evolution of Heuristic miner with the aim of addressing its drawbacks and weaknesses. However, depending on the event log Fodina not always delivers better results of Heuristic miner. Despite that, it is an alternative option that should be taken into account where Heuristic miner fails on building a proper process model, or as mean of comparison.

Process mining algorithms which produce structured model by construction:

- **Inductive Miner** [LFvdA13]: this algorithm uses a divide-and-conquer approach to discover process trees. A process tree is a tree where each leaf is labelled with an activity and each internal node is labelled with a control-flow operator: *sequence*, *exclusive choice*, *non-exclusive choice*, *parallelism*, or *iteration*. Using the *direct follows dependency* between the event type in the log, it firstly creates a directly-follows graph, and secondly use this latter to identify cuts. A cut represent a specific control-flow dependency along which the log can be bisected. The identification of cuts is repeated recursively, starting from the most representative one until no more cuts can

---

\(^3\)http://www.processmining.be/fodina
be identified. Once all cuts are identified and the log split into portions, a
process tree is generated on top of each portion of the log. The algorithm
then applies filters to remove “dangling” directly-follows edges so that the
result is purely a process tree.

- **Evolutionary Tree Miner** [Bvv12]: this is a genetic algorithm that starts
by generating a population of random process trees. At each iteration, it
computes an *overall fitness* value for each tree in the population and applies
mutations to a subset thereof. A mutation is a tree change operation that
adds or modifies nodes. The algorithm iterates until a stop criterion is
fulfilled, and returns the tree with highest overall fitness. Stop criterion
are: maximum execution time or overall fitness threshold reached.

Independently of the degree of structuredness of the output models, none of
the just presented algorithms deliver as output a BPMN model. Indeed, *Alpha-
Miner*, *Inductive Miner* and *Evolutionary Tree Miner* return a Petri net, whilst* 
Heuristic Miner* and *Fodina Miner* give back a Heuristic net. Both Heuristic
and Petri nets are usually less understandable than BPMN models, especially for
business stakeholders, so that conversion tools are used to turn the discovered
net into a BPMN process model. Besides, whilst the conversion PNs \( \rightarrow \) BPMN
is painless, converting from Heuristic Net to BPMN is more difficult and the
procedure can be error prone. Nevertheless, this does not mean discover a BPMN
model, but simply changing the appearance of the output model.

For this reason a special mention needs **BPMNMiner** [CDGBLR16]: the
first mining tool able to output a hierarchical BPMN model, where “hierarchi-
cal” underlines its capability of discovering expanded subprocesses, even nested.
Moreover, it is able to detect boundary events and activity markers. All these
features are of course not reproducible through a simple conversion from a Petri
or Heuristic net to a BPMN model. This makes BPMNMiner the only choice
when we want to obtain a real BPMN model as output.

However, BPMNMiner still relies on the above mining algorithms, in-
deed it firstly identifies in the event log key attributes that can represent
a subprocess (e.g. event types linked to a certain resource), successively it
parts the event log according to the key attributes selected and for each of
them a subprocess will be mined using one of the above mining algorithms.
Finally, it rebuilds the final BPMN model with all the subprocesses and possibly
boundary events, marking (where applicable) activities as loops or multi-instance.

For our evaluation, later discussed, we will use BPMNMiner, despite when
models are flat (i.e. with no subprocesses) there would be no difference between
using BPMNMiner embedding a certain mining algorithm or the mining algorithm
itself, exception made for the activity markers.

2.4 Accuracy and Complexity of Discovered
Models

Considering an event log and the underneath process from which it has been
recorded, the output model got at the end of the mining procedure could
widely differ depending on the discovery algorithm used. Therefore, methods
to evaluate which algorithm produced the best model are needed. The quality
of an automatically discovered process model is generally assessed along four
dimensions: recall (a.k.a. fitness), precision, generalization and complexity.

The first three dimensions can be clustered into accuracy metrics and all them
are measured on a scale of 0 to 1.

- Recall (Fitness): it is the ability of a discovered model to reproduce the
  behavior contained in its event log. There are many methods to evaluate
  this metric, however in this thesis, we use the fitness measure proposed
  in [AvDvdA11], which measures the degree to which each trace in the log
  can be aligned with a trace produced by the model. This approach is
widely accepted as the main recall measurement. A score of 0 indicates the inability of the model to reproduce any behavior (e.g., any trace) reported in its log, while a score of 1 indicates the ability to reproduce all the reported behavior.

- **Precision**: it measures the ability of a discovered model to generate only the behavior recorded in its log. A score of 1 indicates that any trace produced by the model can be found inside its log. In this thesis we use the precision measure defined in [AMC+12], which is based on similar principles as the above fitness measure. Accordingly, after generating an alignment automaton describing the set of executed actions and the set of possible actions, this approach measures precision based on the ratio between the number of executed actions over the number of possible actions. We adopted this approach since it is based on the same core principle of the recall measurement. Recall and precision can be combined into a single F-score, which is the harmonic mean of the two measurements \( \frac{2 \cdot \text{Fitness} \cdot \text{Precision}}{\text{Fitness} + \text{Precision}} \).

- **Generalization**: it measures the ability of a discovered model to produce behavior that is not present in the log but that can be produced by the process under observation. To measure generalization we use 10-fold cross validation [Koh95]. We divide the log into 10 parts, discover a model from 9 parts (i.e., we hold-out 1 part), and we measure fitness of the discovered model against the hold-out part. This is repeated for every possible hold-out part. Generalization is the mean of the fitness values obtained for each hold-out part. A generalization of 1 means that the discovered models produce traces in the observed process, even if those traces are not in the log from which the model was discovered.

**Complexity** quantifies how difficult is to understand a model. It can be estimated through a set of various metrics which report on the structure of the process and its control flows. Several of these metrics have been shown to be (inversely) related to understandability [Men08].

- **Size**: it is the number of nodes of the process model.

- **Control Flow Complexity (CFC)**: it is the sum of the weights of all the split gateways of the process model. The weight of a split gateway is equal to the number of possible process execution combinations after the split. That means: 1 for AND-splits, \( \#\text{outgoing}_i \text{flows} \) for XOR-splits, \( 2\#\text{outgoing}_i \text{flows} - 1 \) for OR-splits.
2.5. PROCESS STRUCTURING

- **Average Connector Degree (ACD)**: it is the average number of outgoing and incoming flows of the gateways of the process model.

- **Maximum Connector Degree (MCD)**: it is the maximum number of outgoing and incoming flows of a gateways of the process model.

- **Coefficient of Network Connectivity (CNC)**: it is the ratio between the number of flows and the number of nodes of the process model.

- **Density**: it is ratio between the number of flows of the process model and the maximum number of flows allowed in any other model with equal number of contained nodes.

- **Structuredness**: it is the ratio between the number of nodes contained in SESE regions and all the nodes of the process model.

- **Separability**: it is the ratio between the number of nodes which are articulation points and the size of the process model. An articulation point in a generic graph is a node that once removed will separate a graph into two distinct disconnected graphs.

Also, a mathematical combination of this set of metrics can generate a unique complexity value [Men08], but for sake of understandability they are usually evaluated separately and rarely altogether. Indeed, for our experiments we will focus only on: *size*, *CFC* and *structuredness*.

### 2.5 Process Structuring

The need for block-structured process models led researchers to investigate approaches for the automated structuring of unstructured process models. In particular, Polyvyanyy et al. [PGD12, PGFW14] propose a technique to transform unstructured process models into behaviourally equivalent structured ones. The approach starts by constructing the Refined Process Structure Tree (RPST) [PVV10] of the input process model.

The RPST of a process model is a tree where each node represent a SESE region of the model, and an edge denotes a containment relation between SESE fragments. Specifically, the children of a SESE fragment (i.e. of a node of the RPST) are the SESE fragments that it directly contains. Fragments at the same level of the tree are disjoint.
Figure 2.14: A process model decomposed in its top level SESE fragments.

Since each SESE fragment maps a portion of the process model, it can be classified according to the characteristics of the mapped portion and the type of its children in the RPST. There are four types of SESE regions:

- **Trivial**: it is the minimal SESE fragment, and all the leaves of the RPST are trivials. The model portion it represents is a single flow between two nodes. The entry of this SESE is the source node of the flow, whilst the exit is the target node of the flow (orange fragments in figure 2.14).

- **Bond**: it is a SESE fragment where all its child fragments share two common gateways, one being the entry node and the other being the exit node of the bond. In other words, a bond consists of a split gateway that is also the entry of two or more of its SESE children in the RPST, all of them converging into a join gateway that is the exit of this bond (green fragment in figure 2.14). Each bond in the RPST must have at least two children.

- **Polygon**: it is a sequence of fragments (sequence of all coloured fragments in figure 2.14).

- **Rigid**: it is any other fragment that is not one of the above mentioned (red fragment in figure 2.14).

A model, whose RPST is made up only of trivials, polygons and bonds (i.e. no rigids) is fully structured. Thus the goal of a block-structuring technique is to replace rigid fragments in the RPST with combinations of trivials, polygons and bonds.

In the structuring technique by Polyvyanyy et al., each rigid fragment is unfolded and an ordering relation graph is generated. This graph is then parsed to construct a modular decomposition tree leading to a hierarchy of components from which a maximally structured version of the original fragment is derived.
through the duplication (where needed) of some nodes (activities in our case) of the graph mapped by the rigid. The technique in [PGFW14] produces a maximally-structured version of any acyclic fragment (and thus of any model), but it does not structure unsound processes nor rigid fragments that contain cycles.

The BPM field is not the only one where the problem of structuring unstructured forms has been studied. Originally, the problem of structuring behavioral models has also been studied in the field of programming, specifically for flowcharts: graphs consisting of tasks (instructions), exclusive split and exclusive join gateways. He observed that unstructuredness is always caused by the presence either of an injection (entry point) or an ejection (exit point) in one of the branches connecting a split gateway to a matching join gateway. Also Oulsnam identified six primitive forms of unstructuredness in flowcharts [Oul82], which combined together could generate any other form of unstructuredness. Later, he proposed an approach to structure these six forms [Oul87]. The approach is based on two rules. The first rule deals with an injection, and pushes the injection after the join gateway, duplicating everything that was originally between the injection and the join. On the other hand, when the unstructuredness is caused by an ejection, the ejection is pushed after the join gateway and an additional conditional block is added to prevent the execution of unnecessary instructions. These two rules are recursively applied to the flowchart, starting from the innermost unstructured form, until no more structuring is possible.

Polyvyanny’s and Oulsnam’s technique are complementary: while Polyvyanny’s technique deals mainly with unstructured acyclic rigidis with parallelism, Oulsnam’s one deals with rigid fragments without parallelism (exclusive gateways only). This observation is a centrepiece of the approach presented in the following section.
Chapter 3

Approach

In this chapter we present our proposed approach to discover maximally structured process models with no loss in accuracy (w.r.t the equivalent unstructured discovered model). Input of this approach is an event log along with a few settings to let the internal algorithms reach the best result in terms of execution time, accuracy and complexity metrics of the maximally structured BPMN output model. How to tune the internal algorithms will be explained in chapter 4, whilst in the following we will focus only on how this approach operate through two macro phases: i) discovery and ii) structure, each implemented within a standalone tool, respectively: BPMNMiner and iBPStruct.

3.1 Discovery with BPMNMiner

In this phase a process model is firstly mined from the input log using an existing process mining algorithm. In addition to discover an initial (unstructured) model, this phase also fixes model correctness issues such as: disconnected nodes (structural issues) and deadlocks (behavioural issues).

![Figure 3.1: Overview of the proposed approach.](image)
3.1 DISCOVERY WITH BPMNMINER

3.1.1 Mining

In this step an initial BPMN model is mined from the input event log. The mining can be performed by any process mining algorithm. However, using a process mining algorithm that already produces a structured process model as output (e.g. Inductive Miner or Evolutionary Tree Miner) would be meaningless, due to the fact the next steps would apply no changes to the discovered model.

Therefore, in this thesis we will refer to two mining algorithms which do not deliver structured models by construction: the Heuristics Miner because of its proven accuracy [WBVB12], and Fodina Miner as second alternative.

3.1.2 Cleaning

This second step is in charge of cleaning structural and behavioural issues of the model mined. This cleaning is achieved via 3 heuristics, but before presenting them, we formally define a process model. This formalism is necessary in order to provide mathematically accurate definitions and explanations of the heuristics and algorithms presented in the following.

**Definition 1** (Process model). A process model is a connected graph $G = (i, o, A, G^+, G^x, F)$, where $A$ is a non-empty set of activities, $i$ is the start event, $o$ is the end event, $G^+$ is the set of AND-gateways, $G^x$ is the set of XOR-gateways, their union is $G \equiv G^+ \cup G^x$, and $F \subseteq (\{i\} \cup A \cup G^+ \cup G^x) \times (\{o\} \cup A \cup G^+ \cup G^x)$ is the set of the sequence flows.

The first heuristic (cf. Fig. 3.2) ensures that a model contains a single start and a single end event, and that every activity in the model is on a path from the start to the end. In case of multiple start or end events, these events are connected via an XOR gateway. In case of activities not on a path from start to end, the heuristic places the activity in parallel with the rest of the process, in such a way that the activity can be skipped and repeated any number of times.

The second heuristic ensures that for every bond, the split and the join gateways are of the same type – both AND or both XOR but not mixed (cf. Fig. 3.2). In the case of an acyclic bond (a bond where all paths go from the entry to the exit gateway), the heuristic matches the exit gateway type with that of entry gateway type. If the bond is cyclic (there is a path from the exit to the entry gateway), the heuristic converts all gateways into XORs.

The third heuristic addresses cases of unsoundness related to quasi-bonds. A quasi-bond is a bond with an injection via a join gateway or an ejection via a
split gateway, along a path connecting the entry and exit gateways of the bond. The heuristic replaces the entry and exit gateways of the quasi-bond as well as the join (split) causing the injection (ejection), with XOR gateways.

After applying recursively these heuristics we obtain as output of this step a structural and behaviour correct model, still unstructured but ready to be structured.

### 3.2 Structure with iBPStruct

Once a process model is provided as input of this second phase, it will be firstly structured and successively passed through a procedure of clone detection and removal. Mining algorithms usually do not produce duplicated activities (i.e. activities with same labels), but in order to structure the process model some duplications could be mandatory. Luckily part of these duplication can be (sometimes) removed after the structuring is completed, this to reduce the size of the final model.

#### 3.2.1 Structuring

The third step of our approach deals with the structuring of the discovered process model by removing injections and ejections. Before discussing this step, we need to formally define the notions of activity path, injection and ejection.

**Definition 2 (Activity Path).** An activity path is a path containing activity nodes only (no gateways), between two gateways. Given two gateways $g_{\text{entry}}$ and $g_{\text{exit}}$ and a sequence of activities $S = \langle a_1, \ldots, a_n \rangle$, there is a path from $g_{\text{entry}}$ to $g_{\text{exit}}$, i.e. $g_{\text{entry}} \rightsquigarrow^S g_{\text{exit}}$ iff $g_{\text{entry}} \rightarrow a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n \rightarrow g_{\text{exit}}$, where $a \rightarrow b$ holds if there is a sequence flow connecting $a$ to $b$. Using the operator $\rightsquigarrow$ we can define:
- The set of all paths of a process model:
  \[ P \triangleq \{(g_1, g_2, S) \in G \times G \times A^* \mid g_1 \sim^S g_2\} \]

- The set of incoming paths of a gateway \( g_x \):
  \[ g_x^\ominus = \{(g_1, g_2, S) \in P \mid g_x = g_2\} \]

- The set of outgoing paths of a gateway \( g_x \):
  \[ g_x^\oslash = \{(g_1, g_2, S) \in P \mid g_x = g_1\} \].

**Definition 3** (Injection). Given four different gateways \( g_1, g_2, g_3, g_4 \), they constitute an injection \( i = (g_1, g_2, g_3, g_4) \) iff \( \exists (S_1, S_2, S_3) \in A^* \times A^* \times A^* \mid g_1 \sim^{S_1} g_2 \land g_2 \sim^{S_2} g_3 \land g_3 \sim^{S_3} g_2 \) (see “before” column in figure 3.3).

**Definition 4** (Ejection). Given four different gateways \( g_1, g_2, g_3, g_4 \), they constitute an ejection \( e = (g_1, g_2, g_3, g_4) \) iff \( \exists (S_1, S_2, S_3) \in A^* \times A^* \times A^* \mid g_1 \sim^{S_1} g_2 \land g_2 \sim^{S_2} g_3 \land g_2 \sim^{S_3} g_4 \) (see “before” column in figure 3.3).

Whenever in a single entry single exit region of a process any injection or ejection is present, that region is a rigid. According to [PVV10], a rigid can be classified on the nature of its embedded injections and ejections as follows:

- **XOR-homogeneous Rigid**: if \( \forall \) injection \( i = (g_1, g_2, g_3, g_4) \) \( \land \forall \) ejection \( e = (g_1, g_2, g_3, g_4) \rightarrow g_1 \in G^x \land g_2 \in G^x \land g_3 \in G^x \land g_4 \in G^x \);

- **AND-homogeneous Rigid**: if \( \forall \) injection \( i = (g_1, g_2, g_3, g_4) \) \( \land \forall \) ejection \( e = (g_1, g_2, g_3, g_4) \rightarrow g_1 \in G^+ \land g_2 \in G^+ \land g_3 \in G^+ \land g_4 \in G^+ \);

- **Heterogeneous Rigid**: in any other case.

Moreover, if an injection or ejection is part of a cycle the rigid is further classified as cyclic, otherwise it is classified as acyclic. Finally if any unsound pattern is detected within the rigid it will be said unsound.
3.2. STRUCTURE WITH IBPSTRUCT

Algorithm 1: Structuring flow

input: RPST $rpytest$

1. Queue $Queue := \text{getLeaves}(rpytest)$;
2. Set $Visited := \emptyset$;
3. while $Queue \neq \emptyset$ do
   4. node := remove($Queue$);
   5. parent := getParent(node);
   6. if isRigid(node) then
      7. if isSoundANDHomogeneous(node) OR isSoundHeterogeneous(node) then
         BPStruct(node);
      else
         EOStruct(node);
   8. Visited := Visited $\cup \{node\}$;
   9. if parent $\not\in$ Visited then
      10. insert($Queue$, parent);

Now we have all ingredients to describe the structuring phase. In this phase, the RPST of the cleaned discovered process model is generated and all its rigids identified. Once all rigids have been identified, the RPST is traversed bottom-up, and each rigid is structured along the way.

Algorithm 1 shows how the RPST is traversed and each node is structured. The algorithm uses a bottom-up traversal strategy implemented via a queue. First, all leaves of the RPST are inserted in the queue. At each step a node from the queue is removed, and structured if it is a rigid. The structuring is performed using BPStruct [PGD12] if the rigid is sound and consists only of AND gateways (sound AND-homogeneous) or a mixture of AND and XOR gateways (sound heterogeneous) – cf. line 8. Otherwise the structuring is performed using an extended version of Oulsnam’s algorithm [Oul87] (line 10) as discussed later. Then the node is marked as visited and if the parent node has not been visited yet, it is added to the queue (cf. line 13). This is repeated until the queue is empty.

We decided to use two different structuring techniques since BPStruct guarantees optimal results when applied on sound AND-homogeneous or heterogeneous rigids only, whilst it produces suboptimal results for acyclic XOR-homogeneous rigids and it fails in case of cyclic XOR-homogeneous or unsound rigids. The structuring of these types of rigids is achieved instead using an extended version of Oulsnam’s algorithm.
3.2. STRUCTURE WITH IBPSTRUCT

Algorithm 2: Push-Down

| input: Injection $i = (g_1, g_2, g_3, g_4)$ |
| input: Set of all Paths $P$ |
| input: Set of all Gateways $G$ |

1 if $g_2 \circ \subseteq \circ g_3$ then
2 \hspace{1em} $g'_2 := \text{copy}(g_2)$;
3 \hspace{1em} $G := G \cup \{g'_2\}$;
4 \hspace{1em} $W := \{(g_4, g_2', S) \in G \times G \times A^* \mid \exists (g_4, g_2, S_x) \in (g_4 \circ \cap \circ g_2)[S_x = S]\}$;
5 \hspace{1em} $P := P \cup W$;
6 \hspace{1em} $P := P \setminus (g_4 \circ \cap \circ g_2)$;
7 \hspace{1em} $D := \{(g_2', g_3, S') \in G \times G \times A^* \mid \exists (g_2, g_3, S) \in (g_2 \circ \cap \circ g_3)[S' = \text{copy}(S)]\}$;
8 \hspace{1em} $P := P \cup D$;
9 \hspace{1em} if $(|g_2'| = 1) \text{ AND } (|\circ g_2| = 1)$ then
10 \hspace{2em} $G := G \setminus \{g_2\}$;
11 \hspace{1em} if $(|g_2'| = 1) \text{ AND } (|\circ g_2'| = 1)$ then
12 \hspace{2em} $G := G \setminus \{g'_2\}$;

Before presenting this latter algorithm, we need to introduce two operators.

The first operator is the push-down operator (see Algorithm 2). Given an Injection $i = (g_1, g_2, g_3, g_4)$, Push-Down($i$) can be applied if $g_2 \circ \subseteq \circ g_3$ (see line 1). The operator removes the input injection in four steps:

i. it creates a copy of $g_2$, namely $g'_2$;

ii. for each path from $g_4$ to $g_2$, it changes the end node of the path from $g_2$ to the new gateway $g'_2$ (lines 5 and 6);

iii. for each path from $g_2$ to $g_3$, it duplicates the path, setting $g'_2$ as the starting node of the path, instead of $g_2$ (line 8);

iv. it removes any of $g_2$ and $g'_2$ if it is a trivial gateway (see Fig. 3.3).

The second operator is the pull-up operator (see Algorithm 3). Given an Ejection $e = (g_1, g_2, g_3, g_4)$, Pull-Up($e$) can be applied if $\circ g_2 \subseteq g_1 \circ$ (see line 1). The operator removes the input ejection in four steps:

i. it creates a copy of $g_2$, namely $g'_2$;

ii. for each path from $g_2$ to $g_4$, it changes the starting node of the path from $g_2$ to the new gateway $g'_2$ (lines 5 and 6);
Algorithm 3: Pull-Up

input: Ejection $e = (g_1, g_2, g_3, g_4)$
input: Set of all Paths $P$
input: Set of all Gateways $G$

1. if $\circ g_2 \subseteq g_1$ then
   2. $g'_2 := \text{copy}(g_2)$;
   3. $G := G \cup \{g'_2\}$;
   4. $W := \{(g'_2, g_4, S) \in G \times G \times A^* | \exists (g_2, g_4, S) \in (g_2 \circ \circ g_4) \mid S_x = S\}$;
   5. $P := P \cup W$;
   6. $P := P \setminus (g_2 \circ \circ g_4)$;
   7. $D := \{(g_1, g'_2, S') \in G \times G \times A^* | \exists (g_1, g_2, S) \in (g_1 \circ \circ g_2) \mid S' = \text{copy}(S)\}$;
   8. $P := P \cup D$;
   9. if $|\circ g_2| = 1$ AND $|\circ g_4| = 1$ then
      10. $G := G \setminus \{g_2\}$;
   11. if $|\circ g'_2| = 1$ AND $|\circ g_2| = 1$ then
      12. $G := G \setminus \{g'_2\}$;

iii. for each path from $g_1$ to $g_2$, it duplicates the path, setting $g'_2$ as the end node of the path, instead of $g_2$ (line 8);
iv. it removes any of $g_2$ and $g'_2$ if it is a trivial gateway (see Fig. 3.3).

While the push-down operator is an adaptation of Oulsnam’s technique [Oul87] in our context, the pull-up operator is a new operator. It can be shown that this pull-up operator preserves trace equivalence but does not preserve weak bisimulation equivalence, because it does not preserve the moment of choice (it may pull a choice to an earlier point). Indeed, referring to figure 3.3, given a generic ejection $e = (g_1, g_2, g_3, g_4)$, the pull-up operator by definition can be applied iff $\circ g_2 \subseteq g_1$, this means $g_2$ can be reached only from $g_1$, and consequentially the only way to reach $g_4$ from $g_2$ is passing through $g_1$.

Considering this latter and that by definition the pull-up operator generates a path $S'_{24} \in g'_2 \circ \circ g_4$ for each previously existing path $S_{24} \in g_2 \circ \circ g_4$ (step ii), and creates a duplicate path $S'_{12} \in g_1 \circ \circ g'_2$ for each previously existing path $S_{12} \in g_1 \circ \circ g_2$ (step iii), it follows that for each concatenation of $S'_{12}$ and $S'_{24}$ existing before the pull-up operator, after the pull-up operator there will exist a concatenation of $S_{12}$ and $S_{24}$ that is its duplicate. Therefore, the nature of the pull-up operator does not introduce nor remove executable traces.

The only drawback of the pull-up operator is that the decision to take the path that will lead to $g_4$ is anticipated at $g_1$, whilst before was at $g_2$ (i.e. earlier point of
Algorithm 4: EOStruct (Extended Oulsnam)

\[ \text{input: Rigid } r \]
\[ \text{input: Boolean } pullup \]

\begin{align*}
\text{do} & \\
\text{Set } I := \text{detectInjections}(r); & \\
\text{Set } E := \emptyset; & \\
\text{if } pullup \text{ AND } \text{isSound}(rigid) \text{ then} & \\
\text{Set } E := \text{detectEjections}(r); & \\
\text{if } I \neq \emptyset \text{ then} & \\
\text{Injection } i := \text{cheapestInjection}(I); & \\
\text{if } E \neq \emptyset \text{ then} & \\
\text{Ejection } e := \text{cheapestEjection}(E); & \\
\text{if } (i \not= \bot) \text{ OR } (e \not= \bot) \text{ then} & \\
\text{if } ((e = \bot) \text{ OR } ((i \not= \bot) \text{ AND } (\text{cost}(i) \leq \text{cost}(e))) \text{ then} & \\
\text{Push-Down}(i); & \\
\text{else} & \\
\text{Pull-Up}(e); & \\
\text{while } I \neq \emptyset \text{ OR } E \neq \emptyset; & \\
\end{align*}

choice). Due to this trade-off, we make the use of the pull-up operator optional as discussed below. We contend that in the context of automated process discovery this is not an issue as the input is a set of traces, and hence trace equivalence is a suitable notion of equivalence.

Algorithm 4 (Extended Oulsnam) shows how the two operators are used to structure a rigid fragment. The inputs of the algorithm are an unstructured rigid and a boolean value to indicate whether the pull-up operator is to be used. First, the algorithm detects every injection on top of which the push-down operator can be applied (see line 2), and if the pull-up is enabled, every ejection on top of which the pull-up can be applied (line 5). Second, it selects the cheapest injection and the cheapest ejection (lines 7 and 9). The cheapest injection (ejection) is the injection (ejection) generating the minimum number of duplicates after a push-down (pull-up). Third, the cheapest among these two is then chosen (line 11) and the corresponding operator is applied. The algorithm iterates over these three steps until no more ejections or injections can be removed, which results in a fully structured or maximally structured rigid. Selecting the cheapest injection or ejection at each step does not ensure that the final model will have the minimum number of duplicates. In order to achieve the latter property, we embed the Extended Oulsnam algorithm inside an $A^*$ search [HNR68], where each state
3.2. STRUCTURE WITH IBPSTRUCT

Figure 3.4: An example application of the $A^*$ search tree with our structuring method.

in the search tree is a transformed version of the initial rigid fragment, and
the cost function associated with each state is defined as $f(s) = g(s) + h(s)$ with
$g(s) = \#duplicates$ and $h(s) = 0$. We set function $h(s)$ to zero since it is not
possible to predict how many duplicates are needed in order to structure a rigid.

Figure 3.4 illustrates an example where a rigid is structured using Algorithm 4 within an $A^*$ search. In this example, the rigid has two injections, i.e.
$i_1 = (g_1, g_2, g_3, g_5)$ and $i_2 = (g_2, g_3, g_4, g_5)$. Assuming $i_2$ is the cheapest of the two
injections (i.e. the size of subprocess $G$ is smaller than the size of subprocess $F$),
if we first remove $i_2$ and then $i_1$ (see Step 1.1 and Step 1.1.1) we will have to
duplicate sub-process $G$ twice. This would not happen if we first removed $i_1$ and
then $i_2$ (see Step 1.2 and Step 1.2.1). The use of an $A^*$ search helps us avoid
these situations since it takes care of exploring the search tree and selecting
the sequence of removals of injections and ejections, that leads to the minimum
number of duplicated elements.

Thankfully to the cleaning step the input process of the structuring step should
be sound. Nevertheless, since this structuring algorithm has been implemented
(as already said) in a standalone tool, which purpose is to structure any kind of
process is given as input, the structuring of unsound rigids is a case we have to
deal to as well.

For this latter type of rigids a specific treatment must be used in order to
achieve a consistent result. Indeed, if a rigid is unsound we automatically disable
the pull-up operator independently of the user setup 4, and we only apply the
push-down operator in order to preserve the behaviour of the split gateways of each quasi-bonds that will be turned into bonds when the rigid will be structured.

Successively, after the structuring procedure has been completed, we match the type of the join gateways of the acyclic bonds with the type of their corresponding split gateways (e.g. if the split is an AND gateway the join will be turned into an AND gateway). In case of cyclic bonds, we turn both split and join gateways into XOR to avoid soundness issues. If multiple bonds share the same join gateway, this is replaced with a chain of gateways, one for each bond, maintaining the original bonds hierarchy. However, due to the fact we disabled the use of the pull-up operator, we cannot guarantee that unsound rigids will be fully structured, hence we cannot guarantee that they will be turned into sound fragments.

3.2.2 Clone Detection

As already mentioned, in order to obtain a maximally structured model, some activities have to be duplicated. This drawback led to the implementation of this last step, which aim is to remove as many cloned activities as possible, in order to reduce the size of the output process model. Unfortunately, most of the times these cloned activities cannot be removed without turning back into the unstructured model. This is the reason why after the detection of the duplicates, we have to properly select which of them can be removed without changing the behaviour of the process and without going back into its unstructured version.

Algorithm 5 shows how we manage to achieve this simplification. Before explaining it we would get the focus on the following properties of a bond RPST node, that will be exploited for the clone removal.

i. A bond must always have at least two children RPST nodes.

ii. If a child of a bond is a trivial, the graphical representation of the trivial is an edge connecting the entry gateway of the bond with the exit gateway of the bond.

iii. If a child of a bond is a rigid, it means the entry and the exit of the rigid match the entry and the exit of the bond.

iv. If a child of a bond is a polygon (i.e. a sequence of RPST nodes), the entry gateway of the first RPST node of the polygon’s sequence is the entry gateway of the bond, and the exit gateway of the last RPST node of the polygon’s sequence is the exit gateway of the bond.
Algorithm 5: Clone Detection and Removal

**input:** RPST \( \text{rpst} \)

1. Set \( \text{Nodes} := \text{getNodes}(\text{rpst}); \)
2. Set \( \text{Bonds} := \emptyset; \)
3. Map \( \text{Codes} := \emptyset; \)

4. \textbf{foreach} \( \text{node} \in \text{Nodes} \) \textbf{do}
   5. \( \text{nodeCode} := \text{generateCanonicalCode(node)}; \)
   6. \( \text{put(Codes, node, nodeCode);} \)
   7. \textbf{if} \( \text{isBond(node)} \) \textbf{then}
      8. \( \text{Bonds} := \text{Bonds} \cup \text{node}; \)

9. \textbf{foreach} \( \text{bond} \in \text{Bonds} \) \textbf{do}
   10. Set \( \text{NodesSharingEntry} := \emptyset; \)
   11. Set \( \text{NodesSharingExit} := \emptyset; \)
   12. RPSTnode \( \text{entry}; \)
   13. RPSTnode \( \text{exit}; \)
   14. Boolean \( \text{changed} := \text{false}; \)
   15. \textbf{foreach} \( \text{child} \in \text{getChildrenNodes(bond)} \) \textbf{do}
      16. \textbf{if} \( \text{isTrivial(child)} \) \textbf{then}
         17. \text{continue;}\)
      18. \textbf{if} \( \text{isPolygon(child)} \) \textbf{then}
         19. \( \text{entry} := \text{getPolygonFirstNode(child)}; \)
         20. \( \text{exit} := \text{getPolygonLastNode(child)}; \)
      21. \textbf{else}
         22. \( \text{entry} := \text{child}; \)
         23. \( \text{exit} := \text{child}; \)
         24. \( \text{NodesSharingEntry} := \text{NodesSharingEntry} \cup \text{entry}; \)
         25. \( \text{NodesSharingExit} := \text{NodesSharingExit} \cup \text{exit}; \)
      26. \( \text{changed} = \text{checkClonesAndRemove(NodesSharingExit, Codes)}; \)
      27. \textbf{if} \( \text{NOT changed} \) \textbf{then}
         28. \( \text{changed} = \text{checkClonesAndRemove(NodesSharingEntry, Codes)}; \)
v. A child of a bond cannot be a bond. Otherwise the child bond would share the entry and the exit gateways of the parent bond, and therefore the children of the child bond would actually be children of the parent bond.

Input of the clone detection and removal algorithm is the RPST of the process model (after it has been structured). Firstly, the algorithm retrieves all the nodes of the RPST (line 1). Then, it associates to each RPST node its Canonical Code, keeping memory of these couples using a map (respectively line 5 and 6). The Canonical Code is a unique code computed on an RPST node, two RPST node are clones iff they have the same Canonical Code [UDGBLR11]. While generating the codes, the algorithm also saves in a set the RPST nodes that are bonds (line 8). Successively, the children of each bond are evaluated in order to find out which of them share the entry and/or the exit of the parent bond. If the child is a trivial, it is discarded, because it will contain no activities (i.e. no clones). If it is a polygon the first and the last RPST nodes of the polygon’s sequence are selected because sharing respectively the entry and the exit of the analysed bond (line 19 and 20). If the child is not a trivial nor a polygon, it could be a rigid in case the structuring wasn’t totally achieved. In this latter case, the rigid shares both entry and exit with the parent bond, so it will be selected for either the NodesSharingEntry set (line 22) and the NodesSharingExit set (line 23). Finally, all the children identified as nodes sharing the exit of their parent bond (NodesSharingExit set) are input to a function which will check if they are all clones (line 26), and if so, all them will be removed except for one, whose entry will merge the entries of all its twin clones in order to preserve the behaviour of the process. The removal can be performed iff all the nodes in NodesSharingExit are clones, otherwise any removal could entail a change in the process behaviour. If this removal is successful, the analysis will continue with the next bond, indeed, no further changes can be applied to the actual one, due to the fact its internal structure has already been modified. Instead if the removal is unsuccessful (i.e. there exist one or more elements in NodesSharingExit that is not clone of the others elements), the algorithm looks for clones which share the entry of the bond (line 28). Also in this case the function checkClonesAndRemove will assess whether all the elements in NodesSharingEntry have exactly the same Canonical Code, and if so, all these elements would be removed from the model, except for one, whose exit will merge the exits of all its twin clones in order to preserve the behaviour of the process. Once all the bonds have been evaluated, if any change occurred, a new RPST is generated from the new model and the clone detection and removal algorithm is run again, that until no further removal
can be applied.

The reason why the clones are sought near the entry and the exit of the bonds is due to the way the pull-up and the push-down operators operate. The former accumulates clones near the entry of a bond, whilst the latter near the exit of a bond. Whenever the pull-up operator is disabled, no clones would be detected sharing the entry of a bond, because none of them could have been generated during the structuring phase.

Here follows an example of structuring, clone detection and removal. Figure 3.5 shows the input model for the structuring. It is a process model mined using Heuristic Miner as embedded discovery algorithm in BPMNMiner. The model is unstructured, but sound. After the structuring phase it contains three bonds which are clones, figure 3.6 highlights the clones in orange and green. The two different colors differentiate the containing bond of each clones. Indeed, the green one is the last element of one of the two polygon children of the top level bond. Whilst the last element of the other polygon (child of the top level bond) is the parent bond of two other polygons, which last elements are the two orange bonds. Also, these latter share the exit of the parent bond of their parent polygons. Therefore, after the first run of the clone detection and removal algorithm, they will be detected and one of them removed. After this first removal, the resulting model is the one in figure 3.7. At this point the remaining clones are both highlighted in green because now they are both last elements of two polygons, whose parent is the top level bond. This happened as result of the first removal. Indeed, the survived orange bond changed its parent polygon. Since during the last execution of the clone detection and removal algorithm a change has been applied, it will be run again once more to check if other clones are detected. After this second run the two green bonds will be selected as clones sharing the same exit, and removed. Finally, a third time the algorithm will run, but this last time no changes will be applied and the tool will output the resulting process model (figure 3.8).

3.3 Complexity Analysis

For the evaluation of the complexity, we leave BPMNMiner out of the computation, focusing only on the added value of iBPStruct.

About the structuring, we have to consider the complexity of the push-down and the pull-up operators. For both the complexity is linear on the number of
3.3. COMPLEXITY ANALYSIS

Figure 3.5: A process model mined using Heuristic Miner.

Figure 3.6: Process model in figure 3.5 after the Structuring.

Figure 3.7: Process model in figure 3.6 after the first clone removal.
activity paths to be duplicated when structuring an injection or ejection, i.e. $O(|g_2 \cap \circ g_3|)$. This is bounded by $O(n^2)$, where $n$ is the number of nodes in the model. The complexity of the Extended Oulsnam algorithm is linear on the number of injections and ejections, which is $O\left(\binom{n}{2}\right)$ where $g$ is the number of gateways, which is bounded by the number of nodes $n$. Hence, $O\left(\binom{n}{2}\right) + O(n^2) \approx O(n^4)$. Finally, the complexity of $A^*$ is $O(b^n)$ where $b$ is the branching factor and $q$ is the depth of the solution. In our case the branching factor is the number of injections and ejections, and so is the depth of the solution. Hence the complexity of our method is $O(n^4) \cdot O(n^4) \approx O(n^{n^4})$.

Since the complexity of the $A^*$ can highly effect the time performance of our approach, other search algorithms have been implemented along with the $A^*$, such as: Depth-first algorithm and a Limited$A^*$ (restrictions are applied on the number of children the algorithm can generated each time it explore a state, i.e. limitation of the branching factor).

About the clone detection, the complexity for the computation of the Canonical Code is linear on the number of activities within the RPST node, so that it is bounded by the number of nodes of the input model, resulting in: $O(n)$. The Canonical Code is computed for each RPST node, the number of RPST node is bounded by the number of nodes of the model, therefore we obtain: $n \cdot O(n) \approx O(n^2)$. The function checkClonesAndRemove is linear on the size of the input set, and in the worst case it is called twice for each bond. Considering that both the size of the input set and the number of bonds can be bounded by the number of nodes of the model, it follows that the complexity checkClonesAndRemove is $O(n)$, but it is executed twice (in the worst case) and it becomes $2 \cdot O(n) \approx O(n)$. Also, adding the iteration on each bond we finally obtain $n \cdot O(n) \approx O(n^2)$. This latter has to be added to the complexity of the computation of the Canonical Code, turning into: $O(n^2) + O(n^2) \approx O(n^2)$. Besides, the algorithm for clone detection and removal can be iterated several times, but not more than the number of clones, that is bounded by the number of nodes of the model.
model, and finally we obtain: $n \ast O(n^2) \approx O(n^3)$.

The total complexity of iBPStruct is the sum of the complexity of the structuring and the complexity of the clone detection and removal, so that at the end it is: $O(n^n) + O(n^3) \approx O(n^n)$. 
Chapter 4

Implementation and Deployment

Our proposed approach has been implemented and deployed in two popular and important developing environments in the context of BPM and process mining. One is Apromore, which is also a process repository, and the second is ProM.

In this chapter we will introduce this two platforms and we will show how to tune and use the developed tool in order to get the best results. Further, since our approach is not simply a tool but a tool-chain, we will give a glance to each of the blocks composing this tool-chain, starting with iBPStruct (the core of our approach) and ending with BPMNMiner and the embedding of iBPStruct into it, passing through an intermediate state that is the Structured Miner [ACD\textsuperscript{+}16].

Finally, we also implemented a complexity calculator: a useful tool that allows the user to measure any complexity metric on top of a BPMN or CPF process model. This latter was necessary to evaluate the quality of our approach, as shown in the next chapter.

4.1 Apromore

Apromore (http://apromore.qut.edu.au/) is a web platform that provides a wide set of services useful for the storage of large collections of process models and process model analysis. Initially, it was born as process models repository, but its extensible nature allowed it to grow as developing environment for OSGi-plugins which operate mainly on process models and event logs. Apromore architecture is base on three components (see figure 4.1):

- the manager, which embeds the database management system for the interfacing of the repository. It includes all the computational aspects such as API to securely access, elaborate and save the process models. Also
it contains the logic of the OSGi-plugins deployed in Apromore. For its features, the manager is the most important block, though for users not involved in developing of OSGi-plugin it is a completely invisible layer.

- the **portal**, which provides the user interface for the interaction with the repository. It displays the content of the database in a file-system like view, organizing the models into folders (figure 4.2). Also it allows to have different tabs, which are mostly exploited to show the results of the OSGi-plugins. Indeed, the majority of the OSGi-plugins developed so far for Apromore are accessible from the portal (through a drop-down menu labelled: Plugins), getting as input the selected process models.

- the **editor**: which is in charge to display the process models (as shown in figure 4.3). It offers editing functionalities for the modelling of process models through several modelling languages. Further, the editor embeds those OSGi-plugins that need to be execute on the process model, such as graphical plugins, animation plugins or even logic-plugins which results would be better understood when shown along with the input model.

These three main components interact together to provide a flexible infrastructure upon which new algorithms and techniques can be implemented and deployed as OSGi-plugins. These plugins can be classified according to which component they are going to live in. Therefore we have three classes of plugin:

- **portal-plugins**: these are all those plugins which want as input one or
4.1. APROMORE

Figure 4.2: Apromore portal interface.

Figure 4.3: Apromore editor interface.
more process models and/or event logs. They live within the portal and display their results in a new tab or in case the output is just a new process model, they save it inside the repository.

- **editor-plugins**: these are all those plugins which need as input one and only one process model, plus possibly an event log. They mostly deal with graphical aspects and have to output a change on the displayed process model in the editor.

- **logic-plugins**: these plugins contain the logic core of the previous mentioned, indeed both portal and editor-plugins should not contain any logic except for the one necessary to interface with the portal and the editor components. Whilst they should rely on logic-plugins deployed inside the manager for what concern the computational duty.

Finally, this architecture delivers the following unique feature: the possibility of loading and unloading plugins without the need to restart the production server. Additionally, thanks to the use of OGSi, Apromore is capable of supporting multiple versions of the same plugin at the same time. This feature not only guarantee retrocompatibility among plugins, but also prevents the unfulfillment of dependencies when a new version of a plugin is release, since other plugins may be using functionality no longer supported in the new version of the plugin.

Apromore is an open-source project and its source code can be downloaded trough GitHub at [https://github.com/apromore](https://github.com/apromore).

### 4.2 ProM

ProM is a desktop Java application for process mining. Similarly to Apromore, it allows developers to implement and deploy plugins within its environment. Though it do not interface with any repository and the inputs for each plugin have to be loaded into the application everytime it is restarted. Further, each output is not saved automatically anywhere, therefore it is up to the user export any kind of result obtain after the running of a plugin.

Nowadays it is more than ten years ProM is used by researchers and practitioners of the field of BPM and process mining, and it counts over 300 plugins. Like Apromore it comes as open-source project and it is free downloadable at: [www.promtools.org/](http://www.promtools.org/).
4.2. PROM

Figure 4.4: ProM data tab.

Figure 4.5: ProM execute tab.
Figure 4.4 and 4.5 show respectively a view of ProM data tab containing the loaded objects that can be used as input for the plugins, and ProM execute tab where is possible select the plugin and its inputs.

4.3 iBPStruct

iBPStruct, as already said, is the core of our approach. It implements the logic to structure an input BPMN process model and, successively, it detects clone activities and try to remove them applying the algorithms explained in chapter 3.

In Apromore, iBPStruct exists as logic, portal and editor-plugin. The logic-plugin has been developed as an OSGi-bundle, it contains the implementation of the algorithms and it has been deployed inside the manager.

The portal-plugin provides an user interface to tune the input parameters for the algorithms. In order to use iBPStruct, we have firstly to select the process we want to structure from those available in the repository (remembering the reader, at any time any process can be uploaded to Apromore). Figure 4.6(a) shows the selected process is \texttt{a161\_2770\_hm}. Secondly we have to open the \textit{Plugins} drop-down menu and click on \textit{Structure}. This will trigger the iBPStruct portal-plugin and its setup window will pop-up as in figure 4.6(b). The input parameters are the following:

- **Structured Process Name**: the output process name when it will be saved into Apromore. Already existing names are not allowed, and if any is chosen the plugin would trigger an error.

- **Structuring Policy**: it allows to select which search algorithm has to be used during the structuring. As explained in chapter 3, the $A^*$ algorithm could be computationally heavy and it could take a lot of time before ending. In such cases it can be time bounded, or we can opt for another structuring policy such as: \textit{Depth – First} or \textit{Limited – $A^*$}. Both of them do not ensure the optimal solution, but they are faster than $A^*$, so that we strongly recommend their use when the size of the process model exceeds 50 nodes (with at least 20 of them as gateways). For testing purpose, also the search policy \textit{Breadth – First} has been implemented and it is still an available option, however we discourage its use, because it is not fast nor deliver optimal results.
- **Time Bounded** $A*$: when this flag is set to *yes*, it ensures the search algorithm $A*$ will not run for more than a certain time in minutes (input parameter later introduced). Specifically, if $A*$ is selected and time bounded, once the timer is over if any solution was met the best of them is picked. Otherwise, the structuring policy automatically switches to $Limited - A*$, which continues the search for a solution. However, also this latter will execute for a maximum time equal to the one specified as time bound for $A*$. Finally, if neither $Limited - A*$ converges to a solution within the timer ending, the search algorithm automatically switches to $Depth - First$, and it will converge before the timer expires for the third time or it will return the best maximally structured model produced so far.

- **Allow Pull-Up Operator**: this is the flag that enable or disable the pull-up operator.

- **Force Structuring**: this flag allows to bypass $BPStruct$ and structure each rigid of the process with the Extended Oulsnam Algorithm. It was implemented because $BPStruct$ always preserve the behaviour of the input process, but because of this restriction it sometimes could not achieve a fully structured rigid. The user can choose whether keep the behaviour leaving this flag off, or prefer a fully structured rigid setting this flag on.

- **Time Bound in Minutes**: this slider is used to set the time bound in minutes when $A*$ is time bounded.

- **Branching Factor**: this slider allows to set-up the $Limited - A*$, its value corresponds to the number of children the $Limited - A*$ keeps as successors of each state analysed. Whether a greater number of successors is generated after the analysis of a state of the search tree, the exceeding will be discarded as follows: the successors are firstly ordered on the value of their $f(s)$ and then discarded starting from those the highest value of $f(s)$, until a number of successors equal to this parameter is reached.

- **Max Equal State To Keep**: this slider is used to prune the search tree while running $Limited - A*$. The search algorithm keeps only the selected number of states for each value of $f(s)$. For instance, if while exploring the search tree we met 51 states with the same $f(s)$, but this parameter is set at 50, one of the 51 states is discarded. Specifically the one closest to the root of the search tree.
4.3. IBPSTRUCT

(a) iBPStruct portal-plugin. (b) iBPStruct setup window.

Figure 4.6: iBPStruct interface.

- **Max Solutions to Reach:** this slider set a termination criterion for the search algorithm *Limited − A* . It stops whenever it met a number of solutions equal to this parameter, regardless the quality of the solutions.

- **Max Depth to Reach:** this slider set a termination criterion for the search algorithm *Breadth − First*. It stops as soon as it has totally explored the search tree till the depth specified with this parameter. Since each level of depth corresponds to the removal of an injection or ejection, this can be used to assess whether a rigid can be fully structured removing only a certain number of injection or ejection.

After each input parameter is set, we can structure the process model selected simply clicking *Structure* at the bottom of the setting window. Once the algorithm ends, a new model will be saved into the repository named as chosen
in first input parameter. Double-clicking on this latter will show it in the editor.

The editor-plugin of iBPStruct is accessible from the editor view. As the portal-plugin, it contacts the logic-plugin inside the manager for the computation duty, but in this case the input is the model loaded inside the editor at the moment of launching iBPStruct. Further, the outputs is directly shown in the editor without saving it in Apromore database. This means it is up to the user save the resulting process model, or edit it, or discard it.

The source code of all the iBPStruct plugins is embedded into the Apromore project and available for download at the same GitHub repository.

Also iBPStruct is implemented in ProM as a plugin named: Structure BPMN Diagram (see figure 4.7). It receives as input a BPMN Diagram object (pre-loaded into ProM) and it gives as output another BPMN Diagram object labelled Structured Diagram.

### 4.4 Structured Miner

The current version of iBPStruct is able to structure any kind of BPMN process models, flat and hierarchical. Also it can deal with boundary events, activity
markers and swimlanes. However, its first implementation was only capable to structure flat models, therefore it could not be used as chain with BPMNMiner. For such reason, the primitive version of the approach introduced in this thesis was *Structured Miner*, which embeds the first version of iBPStruct. At moment of writing, Structured Miner can be found only on ProM, or as standalone Java application available for download at [http://apromore.org/platform/tools](http://apromore.org/platform/tools), coming with a user guide.

In ProM, this mining tool receives in input an event log (pre-loaded into ProM), successively it asks the user to select the bottom mining algorithm (embedded are Heuristic Miner and Fodina Miner), and to set-up iBPStruct. As result it outputs a maximally structured BPMN process model as BPMN Diagram object. This primitive implementation of our approach generates only flat process models (i.e. containing no subprocesses), and it was used for the evaluation in [ACD+16].

### 4.5 BPMNMiner 2.0

*BPMNMiner 2.0* is the final implementation of the approach presented in chapter 3. It is the chain tool embedding BPMNMiner(1.0) [CDGBLR16], used for the discovery phase, and the extension of the first implementation of iBPStruct, used for the structuring phase. This latter, as already said, is able to deal with hierarchical BPMN process models, simply structuring each subprocess separately and then rebuilding the whole process.

This tool is available in Apromore and in ProM, in the former is a portal-plugin named: *Discover Process Model* (still in the same drop-down menu of iBPStruct) and its logic is deployed inside the manager as logic-plugin. Whilst in ProM it is a plugin named: *BPMNMiner Wrapper*. For both plugins the input is an event log and the output is a hierarchical BPMN model, whether the models contain subprocesses, otherwise the output would still be a flat BPMN model.

A tutorial video on how to use BPMNMiner 2.0 in Apromore is accessible at: [https://www.youtube.com/watch?v=eb0k2R02PQ8](https://www.youtube.com/watch?v=eb0k2R02PQ8).
4.6 Complexity Calculator for BPMN and CPF Models

The Complexity Calculator, along with iBPStruct, has been implemented as logic, portal and editor-plugin. This tool can be triggered through the menu item: Measure in the Plugins drop-down menu, as shown in figure 4.8. It takes as input the selected process model in both its native and CPF representation. Then, it will compute all the complexity metrics presented in section 2.4 on top of the CPF representation of the model. Also, the tool outputs the number of rigids and bonds found in the process model. Successively if the native representation of the model is BPMN it will also compute the complexity metrics on the BPMN model. The output in the portal is a tab containing the values of the metrics for the BPMN model (if available) and the CPF model (see figure 4.9).

About the editor-plugin of the Complexity Calculator, it can be accessed through the editor interface after a process model has been opened in it. However, this second implementation of the plugin works only on top of BPMN models, this because no graphical representation is available for CPF models. The output is a pop-up window showing the value of each metric.

Finally, the Complexity Calculator has been implemented also as ProM
Compliance with Complex BPMN and CPF Models

Figure 4.9: Complexity Calculator output.

Plugin, but in this case it is able to run only on BPMN process models. Indeed, ProM do not implements CPF objects but only BPMN Diagram objects.

The source code of this tool is accessible through the Apromore project.
Chapter 5

Evaluation and Results

We conducted a series of experiments to evaluate the accuracy and the complexity of our discovery approach compared to those of methods that discover block-structured process models by construction. From this category we selected as representative methods: Inductive Miner (IM) and Evolutionary Tree Miner (ETM), and we compared the results with our approach on top of Heuristic Miner (HM) and Fodina Miner (FO).

We measured the accuracy of the discovered process models using fitness, precision, F-score and generalization (10-fold). Whilst for the complexity we decided to assess it via size, CFC and structuredness.

All the experiments were done on an Intel dual-core i5-3337U 1.80Ghz with 12GB of RAM running JVM 7 with 8GB of heap, except for the experiments regarding ETM. Indeed, since ETM is computationally very expensive because of its genetic algorithm, a little advantage was mandatory to let it achieve a proper result. For the ETM experiments we relied on a 6-core Xeon E5-1650 3.50Ghz with 128GB of RAM running JVM 7 with 40GB of heap. Further, we also decided to set as stop criterion of ETM a time-bound of 30 minutes. As the reader can expect, we took into consideration this drawback of ETM at the moment of the final evaluation of the result obtained.

5.1 Setup and Dataset

To run our experiments we generated three sets of logs using the ProM plugin “Generate Event Log from Petri Net” [VBDWVB12]. This plugin takes as input a process model in PNML format and generates a distinct log trace for each possible execution sequence in the model. The resulting logs range from 4,111 to
201,758 total events (avg. 49,580) with 3 to 4,235 distinct traces (avg. 137).

The first set (591 Petri nets) was obtained from the SAP R/3 collection, SAP’s reference model to customize their R/3 ERP product [CK97]. The log-generator plugin was only able to parse 545 out of 591 models, running into out-of-memory exceptions for the others. The second set (54 Workflow nets\(^1\)) was obtained from a collection of sound and unstructured models extracted from the IBM BIT collection [DLM\(^+\)12]. The BIT collection is a publicly-available set of process models in financial services, telecommunication and other domains, gathered from IBM’s consultancy practice [FFK\(^+\)11]. The third set contains 20 artificial models, which we created to test our method with more complex forms of unstructuredness, not observed in the two real-life collections. These are:

i. rigids that embed AND-gateway bonds;

ii. rigids containing a large number of XOR gateways (> 5);

iii. rigids nested into rigids;

iv. rigids being the root node of the RPST of the model.

Out of these 619 logs we only selected those for which HM and/or FO discovered an unstructured model (either sound or unsound), as our approach does not add any value if the discovered model is already structured. Despite the process models delivered by HM and FO are most of the times similar (if not identical), sometimes they delivered models with different characteristics. For such reason we had to separate and classify the models twice: once for HM and once for FO.

Regarding the processes discovered by Heuristic Miner, only 126 were useful for our analysis. Of these, 61 came from SAP, 45 from IBM and 20 were synthetic. From the models discovered with HM, we identified 79 sound models, 31 partially unsound models (i.e. models for which there is at least one complete trace from the start event to the end event), and 16 totally unsound models (i.e. models whose traces always deadlock). A taxonomy of the datasets used for the experiments with HM is shown in figure 5.1.

Instead, for what concerns Fodina Miner, 137 logs could be used for our analysis. Of these, 72 came from SAP, 45 from IBM and 20 were synthetic. From the discovered models with FO, we identified 84 sound models, 52 partially unsound models, and only 1 totally unsound model. A taxonomy of the datasets used for the experiments with FO is shown in figure 5.2.

\(^1\)This collection originally counted 59 models, but we discarded five duplicates.
5.1. SETUP AND DATASET

Figure 5.1: Taxonomy of models discovered by Heuristic Miner from the logs.

Figure 5.2: Taxonomy of models discovered by Fodina Miner from the logs.
5.2 Results

In this section we will discuss the results of our experiments. Firstly we will focus on those obtained using Heuristic Miner as discovery algorithm embedded within BPMNMiner 2.0, because it delivered the best output models in terms of overall accuracy and complexity. Secondly we will give a quick overview on the results got from Fodina Miner, to assess the consistence of our approach, showing that changing the mining algorithm improvements are still delivered.

5.2.1 Heuristic Miner

Tables 5.1 and 5.2 report the average value and the standard deviation respectively for accuracy metrics and complexity metrics of the experiments ran with Heuristic Miner on logs generated from real-life data. Similarly, tables 5.3 and 5.4 displays the results got from artificial data still using Heuristic Miner as mining algorithm embedded in BPMNMiner. We decided to split the evaluation of real-life and artificial data in order to better show what happens in case the unstructured forms become more complex (instance of artificial data).

When HM generates sound models its output is already of high accuracy along fitness, precision and generalization, with a marginal standard deviation. In this case, our approach only improves the structuredness of the models, at the cost of a minor increase in size and CFC, due to the duplication introduced by the structuring.

IM instead, despite having similar high values of fitness and generalization, it loses in precision with an average of 0.69 and a high standard deviation, meaning that the actual precision may be much better or worse depending on the specific log used. As expected, these models are structured by construction, but CFC still remains higher than that of HM (and its structured variant of BPMNMiner) due to IM’s tendency to generate flower models (which is also the cause of low precision). Finally, the quality of the models discovered by ETM ranks in-between that of IM and HM both in terms of accuracy and complexity, at the price of sensibly longer execution times and the need of a more powerful hardware.

The improvement of our method on top of HM is more evident when the latter discovers unsound models. Here HM’s accuracy dramatically worsen compared to IM and ETM. For example, in the case of partially unsound models, on average fitness is 0.69 for HM vs. 0.98 for IM on real-life data, and 0.61 vs. 1 on artificial data, while for totally unsound models, fitness and precision for HM cannot even be measured. Our approach does not only notably increases structuredness (i.e.
To illustrate how our approach outperforms IM, figure 5.3 shows the BPMN model generated by IM from an event log of the SAP R/3 collection. Also,
5.2. RESULTS

<table>
<thead>
<tr>
<th>Log Class (Class Size)</th>
<th>Discovery Method</th>
<th>Fitness</th>
<th>Precision</th>
<th>F-score</th>
<th>Gen.(10-fold)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sound (16)</td>
<td>IM</td>
<td>1.00 ± 0.01</td>
<td>0.53 ± 0.31</td>
<td>0.64 ± 0.26</td>
<td>1.00 ± 0.01</td>
</tr>
<tr>
<td></td>
<td>ETM</td>
<td>0.89 ± 0.07</td>
<td>0.96 ± 0.05</td>
<td>0.92 ± 0.04</td>
<td>0.89 ± 0.05</td>
</tr>
<tr>
<td></td>
<td>HM</td>
<td>1.00 ± 0.00</td>
<td>1.00 ± 0.00</td>
<td>1.00 ± 0.00</td>
<td>1.00 ± 0.00</td>
</tr>
<tr>
<td></td>
<td>BMHM</td>
<td>1.00 ± 0.00</td>
<td>1.00 ± 0.00</td>
<td>1.00 ± 0.00</td>
<td>1.00 ± 0.00</td>
</tr>
<tr>
<td>P. Unsound (4)</td>
<td>IM</td>
<td>1.00 ± 0.00</td>
<td>0.44 ± 0.27</td>
<td>0.56 ± 0.22</td>
<td>1.00 ± 0.00</td>
</tr>
<tr>
<td></td>
<td>ETM</td>
<td>0.83 ± 0.12</td>
<td>0.88 ± 0.09</td>
<td>0.84 ± 0.07</td>
<td>0.78 ± 0.15</td>
</tr>
<tr>
<td></td>
<td>HM</td>
<td>0.61 ± 0.16</td>
<td>0.84 ± 0.06</td>
<td>0.69 ± 0.14</td>
<td>0.61 ± 0.16</td>
</tr>
<tr>
<td></td>
<td>BMHM</td>
<td>0.89 ± 0.13</td>
<td>0.98 ± 0.02</td>
<td>0.93 ± 0.07</td>
<td>0.89 ± 0.13</td>
</tr>
</tbody>
</table>

Table 5.3: Accuracy of models discovered from artificial data with Heuristic Miner.

<table>
<thead>
<tr>
<th>Log Class (Class Size)</th>
<th>Discovery Method</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td>CFC</td>
</tr>
<tr>
<td>Sound (16)</td>
<td>IM</td>
<td>18.7 ± 4.5</td>
</tr>
<tr>
<td></td>
<td>ETM</td>
<td>22.1 ± 7.7</td>
</tr>
<tr>
<td></td>
<td>HM</td>
<td>21.6 ± 5.2</td>
</tr>
<tr>
<td></td>
<td>BMHM</td>
<td>25.1 ± 7.7</td>
</tr>
<tr>
<td>P. Unsound (4)</td>
<td>IM</td>
<td>23.5 ± 10.4</td>
</tr>
<tr>
<td></td>
<td>ETM</td>
<td>25.5 ± 1.5</td>
</tr>
<tr>
<td></td>
<td>HM</td>
<td>27.8 ± 9.1</td>
</tr>
<tr>
<td></td>
<td>BMHM</td>
<td>30.0 ± 12.3</td>
</tr>
</tbody>
</table>

Table 5.4: Complexity of models discovered from artificial data with Heuristic Miner.

<table>
<thead>
<tr>
<th>Pull-Up Operator</th>
<th>Fitness</th>
<th>Precision</th>
<th>F-score</th>
<th>Gen.(10-fold)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disabled</td>
<td>1.00 ± 0.01</td>
<td>1.00 ± 0.01</td>
<td>1.00 ± 0.01</td>
<td>1.00 ± 0.01</td>
</tr>
<tr>
<td>Enabled</td>
<td>1.00 ± 0.01</td>
<td>0.99 ± 0.01</td>
<td>1.00 ± 0.01</td>
<td>1.00 ± 0.01</td>
</tr>
</tbody>
</table>

Table 5.5: Accuracy before and after enabling the Pull-Up operator.

<table>
<thead>
<tr>
<th>Pull-Up Operator</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
</tr>
<tr>
<td>Disabled</td>
<td>30.0 ± 11.1</td>
</tr>
<tr>
<td>Enabled</td>
<td>32.4 ± 14.4</td>
</tr>
</tbody>
</table>

Table 5.6: Complexity before and after enabling the Pull-Up operator.
5.2. RESULTS

Figure 5.3: Discovered model from a SAP R/3 log using Inductive Miner, and its accuracy metrics.

Figure 5.4: Discovered model from a SAP R/3 log using Heuristic Miner, and its accuracy metrics.

Figure 5.4 and 5.5 display the result obtained respectively from HM and BPMN-Miner 2.0 from the same log\(^2\). Along with the models, the accuracy metrics are reported beneath. In this example, the precision of the model produced by IM is low due to the presence of a large “flower-like” structure, which causes overgeneralization. In the model delivered by HM, instead, precision is higher, though fitness and generalization suffer from the model being unsound (highlighted in red). When the HM model is passed through the structuring (mining performed with BPMNNminer 2.0), it fixes the behavioural issues of this model, improving on all accuracy metrics, scoring a perfect 1 for both F-score and generalization.

The negative effects of overgeneralization brought by IM are higher when the models used for generating the logs exhibit even more complex unstructured patterns, such as those introduced in the artificial data (table 5.3). For example, the precision of IM is 0.53 for sound models (with a high standard deviation), as opposed to 1 with HM. Also in these cases, our approach still consistently outperforms IM and ETM, while significantly improving over HM in terms of structuredness (0.3 vs. 1).

In this experiment we disabled the pull-up operator during the structuring phase in order to ensure weak bisimulation equivalence between the model

\(^2\)The original labels are replaced with letters for the sake of compactness.
5.2. RESULTS

Figure 5.5: Discovered model from a SAP R/3 log using BPMNMiner 2.0 on top of Heuristic Miner, and its accuracy metrics.

discovered by HM and the structured one. As a consequence, we could not fully structure 15 models from real-life data, which explains a value of structuredness less than 1 for BPMNMiner in table 5.2. Nevertheless, we decided to run twice the same experiment, the second time enabling the pull-up operator, and in this case all the discovered models were fully structured, at the price of losing weak bisimilarity, but not accuracy. The tables 5.5 and 5.6 display the quality of the 15 models that were not fully structured on the experiment ran with the pull-up operator disabled, compared to their quality after applying the structuring with the pull-up operator enabled.

**Time performance.** Despite having exponential complexity in the worst case scenario, the time iBPStruct took to structure the models used in this evaluation was well within acceptable bounds, taking on average less than one second per model (avg = 894ms, min = 2ms, max = 109s, 95% percentile = 47.65ms).

5.2.2 Fodina Miner

Tables 5.7 and 5.8 report the average value and the standard deviation respectively for accuracy metrics and complexity metrics of the experiments ran with Fodina Miner on logs generated from real-life data. Similarly, tables 5.9 and 5.10 displays the results got from artificial data still using Fodina Miner as mining algorithm embedded in BPMNMiner. Also in this second experiment ran with Fodina Miner we decided to separate the measures done on real-life data from the ones done on artificial data, for the same reason as above. Moreover, since Fodina Miner is less prone than Heuristic Miner on delivering totally unsound process models (only one was present in the dataset of FO), we merged partially unsound and totally unsound processes in the tables showing the accuracy and complexity measures.

As for HM, when FO outputs sound models, their accuracy is high along fitness, precision and generalization, with a marginal standard deviation, so that our approach positively affects only the structuredness of the models, still increasing
5.3 Discussion

Regardless the results got in our experiment, this thesis addresses a simple problem: whenever the underlying model to discover from an event log contains any
5.3. DISCUSSION

Table 5.9: Accuracy of models discovered from artificial data with Fodina Miner.

<table>
<thead>
<tr>
<th>Log Class (Class Size)</th>
<th>Discovery Method</th>
<th>Fitness</th>
<th>Precision</th>
<th>F-score</th>
<th>Gen.(10-fold)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sound (17)</td>
<td>IM</td>
<td>1.00 ± 0.01</td>
<td>0.51 ± 0.30</td>
<td>0.63 ± 0.25</td>
<td>1.00 ± 0.01</td>
</tr>
<tr>
<td></td>
<td>ETM</td>
<td>0.89 ± 0.06</td>
<td>0.96 ± 0.04</td>
<td>0.92 ± 0.04</td>
<td>0.89 ± 0.04</td>
</tr>
<tr>
<td></td>
<td>FO</td>
<td>1.00 ± 0.00</td>
<td>1.00 ± 0.00</td>
<td>1.00 ± 0.00</td>
<td>1.00 ± 0.00</td>
</tr>
<tr>
<td></td>
<td>BM&lt;sub&gt;FO&lt;/sub&gt;</td>
<td>1.00 ± 0.00</td>
<td>1.00 ± 0.00</td>
<td>1.00 ± 0.00</td>
<td>1.00 ± 0.00</td>
</tr>
<tr>
<td>P. Unsound (3)</td>
<td>IM</td>
<td>1.00 ± 0.00</td>
<td>0.49 ± 0.30</td>
<td>0.60 ± 0.25</td>
<td>1.00 ± 0.00</td>
</tr>
<tr>
<td></td>
<td>ETM</td>
<td>0.81 ± 0.14</td>
<td>0.85 ± 0.08</td>
<td>0.82 ± 0.06</td>
<td>0.75 ± 0.16</td>
</tr>
<tr>
<td></td>
<td>FO</td>
<td>0.80 ± 0.09</td>
<td>0.93 ± 0.04</td>
<td>0.86 ± 0.07</td>
<td>0.80 ± 0.09</td>
</tr>
<tr>
<td></td>
<td>BM&lt;sub&gt;FO&lt;/sub&gt;</td>
<td>0.94 ± 0.08</td>
<td>0.97 ± 0.02</td>
<td>0.96 ± 0.04</td>
<td>0.94 ± 0.84</td>
</tr>
</tbody>
</table>

Table 5.10: Complexity of models discovered from artificial data with Fodina Miner.

<table>
<thead>
<tr>
<th>Log Class (Class Size)</th>
<th>Discovery Method</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sound (17)</td>
<td>IM</td>
<td>24.7 ± 4.4</td>
</tr>
<tr>
<td></td>
<td>ETM</td>
<td>22.4 ± 7.5</td>
</tr>
<tr>
<td></td>
<td>FO</td>
<td>21.9 ± 5.1</td>
</tr>
<tr>
<td></td>
<td>BM&lt;sub&gt;FO&lt;/sub&gt;</td>
<td>25.5 ± 7.6</td>
</tr>
<tr>
<td>P. Unsound (3)</td>
<td>IM</td>
<td>24.7 ± 11.7</td>
</tr>
<tr>
<td></td>
<td>ETM</td>
<td>24.0 ± 0.0</td>
</tr>
<tr>
<td></td>
<td>FO</td>
<td>27.7 ± 10.3</td>
</tr>
<tr>
<td></td>
<td>BM&lt;sub&gt;FO&lt;/sub&gt;</td>
<td>32.0 ± 14.4</td>
</tr>
</tbody>
</table>

kind of unstructured form, all those mining approaches which tackle mining and structuring concurrently (intrinsically assuming the original model as structured) will never be able to get the real model but only an approximation of it. Moreover, this approximation will be as rougher as more unstructured is the model concealed in the input event log, entailing a loss in precision and/or fitness.

In such cases, the decision to split discovery and structuring turns into a more reliable method due to the fact that during the mining we have no restrictions on the kind of model we want to discover and only successively we manage to transform it into its structured twin.

However, since the proposed approach relies on an underneath mining algorithm (Heuristic Miner rather than Fodina Miner), it must be kept into account that in those instances where the discovery phase leads to a scarcely fitting model or a spaghetti-like model far away from the original one, the structuring phase will not be able to turn it into a structured version of the real process. This is the reason why our approach does not set any constraints regarding the mining algorithm to use during the discovery phase. Indeed, whereas a better mining algorithm than Heuristic Miner (or Fodina Miner) would be developed, it could be exploited for better results. Moreover, if this supposed mining algorithm
would also be able to deal with data variables we could use the standard version of Oulsnam’s structuring algorithm, being able to structure everything without resigning weak bisimilarity.

5.4 Threats to Validity

A potential threat to internal validity is the use of process model complexity metrics as proxies for assessing the understandability of the discovered process models, as opposed to direct human judgement. However, the three chosen complexity metrics (size, CFC and structuredness) have been empirically shown to be highly correlated with perceived understandability and error-proneness [Men08, DLM+12]. Further, while the process models obtained with our method are affected by the individual accuracy (fitness, precision and generalization) of the baseline algorithm used (HM or FM), our approach is independent of these algorithms, and our experiments show that the method significantly improves on structuredness while keeping the same levels of accuracy. In addition, the method can often fix issues related to model correctness.

The evaluation reported above is based on two real-life datasets. This poses a threat to external validity. It should be noted though that these two datasets collect models from a variety of domains, including finance, sales, accounting, logistics, communication and human resources, and that the resulting logs are representative of different characteristics (number of events and number of distinct traces). Moreover, the use of an additional dataset artificially generated allowed us to evaluate our method against a large variety of unstructured model topologies, including some complex ones not observed in the two real-life datasets.
Chapter 6

Conclusion

In this thesis we discussed an alternative method to strike the problem of process mining, whose aim is to discover the model behind a process starting from its event log. Even though over the last years several mining algorithms have been developed to tackle this problem, none of them guarantees the discovery of a process model having the highest accuracy (i.e. fitness, precision, generalization) and the lowest complexity (i.e. size, control flow complexity, structuredness, etc. [Men08]). This entails that certain mining algorithms tend to discover process models having a high accuracy at the price of increasing their complexity.

Starting from this point, we firstly developed and implemented a technique to reduce the complexity of a process model (in terms of structuredness), which is also able to fix possible behavioural issues (i.e. unsoundness). Secondly, we enqueued this structuring tool to already existing mining algorithms known for the high accuracy of their discovered models, but which suffer of high complexity and even unsoundness.

The resulting two-phased method is able to mine a sound and maximally structured process model from an event log wherein a process model is firstly extracted without any structural restriction, trying to achieve the greatest accuracy relying on a precise mining algorithm, and then the output model is transformed into a structured one, fixing (where needed) behavioural issues.

In the second phase, we employed a structuring method that preserves weak bisimilarity (whether the pull-up operator is disabled). However, we experimented with the pull-up operator either enabled and disabled and the results showed that our structuring phase was more effective whenever the pull-up operator was enabled, being able to fully structure a larger number of process models. Considering that, a direction for future work is to explore the option of partially sacrificing weak bisimilarity (while still keeping trace equivalence) to obtain mod-
els with higher structuredness.

Finally, as side effect of the structuring is the injection of duplicates in the process model, and consequentially a growth in its size, we tried to restrain this drawback using a technique of clone detection and removal in order to limit the final size of the model. However, from our experiments we noticed that rarely we could apply the clones removal without getting back to an unstructured form of the process model. Hence, possible future case of research would be the understanding of when it would be better sacrificing a bit of structuredness in order to reduce the global size of the output model.

Nevertheless the side effect brought by the structuring, the experimental results show that our two-phased method leads to a higher F-score than existing methods that discover a structured process model by construction. Further, being the proposed method modular, different discovery algorithms and block-structuring methods can be plugged into it. Despite for our evaluation we used the Heuristics Miner and Fodina Miner for the first phase, and iBPStruct for the second phase, no restrictions were set on the baseline algorithm to use for the mining, neither for the structuring tool. Therefore, in future work, it would be possible experiment with alternative methods for discovering (unstructured) process models to explore alternative trade-offs between model quality metrics, and also improve the structuring phase trying to simplify even further the output model.
Bibliography


