POLITECNICO DI MILANO

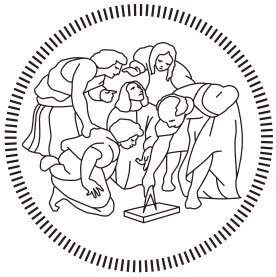
Facoltà di Ingegneria

Scuola di Ingegneria Industriale e dell'Informazione

Dipartimento di Elettronica, Informazione e Bioingegneria

Master of Science in

Computer Science and Engineering



A novel method to consider multiple paths in Decision Points Analysis

Advisor: PROF. MATTEO MATTEUCCI

Co-advisor: DOTT. PIETRO PORTOLANI

Master Graduation Thesis by:

DIEGO SAVOIA

Student Id n. 944508

Academic Year 2021-2022

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

This template has been adapted by Emanuele Mason, Andrea Cominola and Daniela Anghileri: *A template for master thesis at DEIB*, June 2015. This version of the template has been later adapted by Marco Cannici, March 2018.

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Matteo Matteucci, for supervising this thesis and for his valuable guidance. A special thanks to Dott. Pietro Portolani for his assistance and contribution throughout the duration of this work.

I am deeply grateful to my family for their constant support, and in particular to my parents: without them this would not have been possible. Lastly, many thanks to the colleagues with whom I shared my university career, and to the friends who stayed with me during these years.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The analysis of a business process can provide valuable insights which can be exploited in order to achieve a better understanding of the process itself. Process Mining techniques allow to automatically extract a visual model of the process starting from recorded sequences of activities, called Event Logs. The extracted model can then be enhanced with logical conditions constraining the executions of process instances: this is the goal of Decision Mining. More precisely, its Decision Points Analysis subfield aims at augmenting places in a Petri Net model with guards steering the execution of transitions. These guards are usually discovered through machine learning approaches, such as Decision Trees, whose training sets are properly created by retrieving the path followed in the model by each trace. Due to the presence of invisible transitions inside the Petri Net, a trace may correspond to different consistent paths in the model. State-of-the-art approaches only consider a single path, hence possibly losing information. In this thesis, we propose a method which considers multiple paths that a process instance can follow inside the Petri Net model, exploiting more knowledge from the Event Log and consequently obtaining more precise guards. We tested the method both on a synthetic and on a real-life process, showing the improvements with respect to current techniques.

SOMMARIO

L'analisi di un processo aziendale è in grado di offrire preziose informazioni che possono essere sfruttate per ottenere una migliore comprensione del processo stesso. Le tecniche di Process Mining permettono di estrarre automaticamente un modello visivo del processo partendo da sequenze di attività registrate, ovvero i log degli eventi. Successivamente, il modello ottenuto può essere valorizzato con delle condizioni logiche che vincolano le esecuzioni delle istanze di processo: questo è l'obiettivo del Decision Mining. Più precisamente, il sottocampo della Decision Points Analysis mira ad estendere i posti in una rete di Petri con delle guardie che guidano l'esecuzione delle transizioni. Queste guardie sono solitamente ottenute attraverso metodi di machine learning, come i Decision Tree, i cui training set vengono adeguatamente creati risalendo al percorso che ogni traccia segue nel modello. A causa della presenza di transizioni invisibili all'interno della rete di Petri, una traccia potrebbe corrispondere a diversi percorsi nel modello coerenti tra loro. Gli approcci allo stato dell'arte considerano solamente un percorso, perdendo dunque informazioni. In questa tesi proponiamo un metodo in grado di considerare molteplici percorsi che un'istanza di processo può seguire all'interno del modello di rete di Petri, sfruttando maggiormente la conoscenza nel log degli eventi e ottenendo pertanto guardie più accurate. Abbiamo sperimentato il metodo sia su un processo sintetico che su uno reale, mostrando i miglioramenti rispetto alle tecniche attuali.

1 INTRODUCTION

The ability to extract useful information from data has become increasingly widespread [1], thanks to the improvements in hardware and software, allowing for higher computational power and better resource management, but also to the availability of data itself. Data mining and machine learning techniques help in dealing with such massive information, finding hidden patterns and deriving new knowledge from the existing one.

Organizations and businesses often deal with vast amount of data, which can be used to improve the value of the company. The field of *Process Mining* has gained importance over recent years, having the goal of exploiting all those data to analyze and enhance business processes [2]. Indeed, the knowledge about a process is usually recorded by the company inside the so-called *Event Log*. This contains all the information about the process executions: the activities that have been performed, the resources that carried them out, timestamps, and other pieces of data related to the process.

Machine learning comes into play in the *Decision Mining* sub-field of Process Mining: its scope is to extract the rules that steer the decisions inside a process. Indeed, decision making represents one of the most important assets inside an organization, and exploiting machine learning methods to enrich business processes allows to better characterize them, hence increasing their value. Understanding the decisions that led to specific outcomes in a business process can be of crucial importance in order to analyze existing process executions, but also to predict which decisions will be taken in future instances.

There exist various tools and notations to visualize a process, such as *BPMN*, *Process Trees* and *Petri Nets*. They are able to formally represent business processes in order to create a rigorous model which can be used both to get a broader view of the process and to gain new knowledge. In general, a process model is a directed graph composed of a set of nodes and a set of arcs: nodes usually represent activities that are part of the process, or particular states of the process; arcs represent paths that can be followed by the process.

A few approaches have been developed among the Decision Mining field, in which rules are extracted by following the paths in the process model reflecting the sequences of events recorded in the Event Log. Nonetheless, they do not consider all the possible paths in the model, stopping at ambiguous nodes or exploiting only the optimal path. Therefore, these methods suffer from information loss, meaning that a lot of data from the Event Log is not

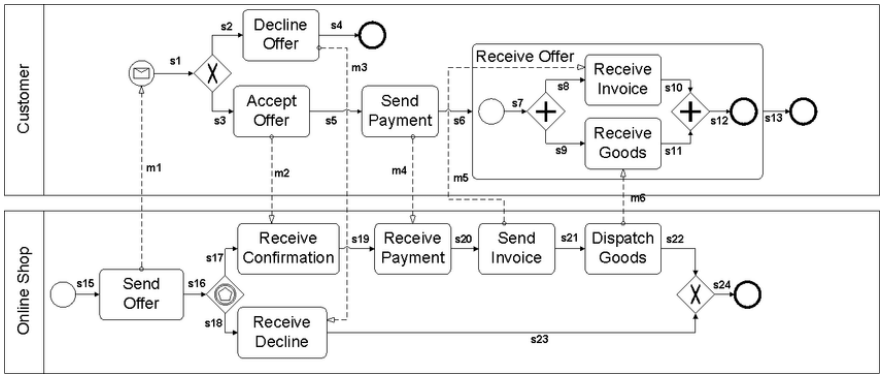


Figure 1.1: Example of business process showing a purchase of an item on an online shop by a customer [3]. The process is represented using a BPMN diagram, in which nodes with activity names represent the events in the process, while arcs represent the possible paths that can be followed. Special rhomboid nodes allows to split the process execution between branches; according to the particular node, all the outgoing branches are followed or only one.

exploited. These approaches rely on Petri Nets in order to model business processes: a Petri Net is a bipartite directed graph with *places* and *transitions* as nodes. A place represents a process state, while a transition represents an activity. The Petri Net model, along with the process Event Log, are used in order to extract the rules governing the choices at decision points, i.e., those places having more than one outgoing arc.

In this thesis, we propose a novel method which better exploits data coming from the Event Log of a business process, resulting in improved rules and, consequently, finer decision-making. This is achieved by considering more than one path among the ones that a process instance could follow inside the Petri Net model. The results obtained using the method we propose support this idea: analyzing the same Event Log and Petri Net model, our method finds rules with higher accuracy and it is also able to discover rules for decision points which previous approaches were not able to find. We also present alternative state-of-the-art approaches both for rules discovering and for rules simplification, which we implemented and often modified in order to better suit the Decision Mining context.

The thesis outline is the following: Chapter 2 presents the state-of-the-art approaches which deal with this problem, along with the background needed to better understand the setting and related works tackling Decision Points Analysis and rules pruning; Chapter 3 contains an in-depth explanation of the method proposed by this work; Chapter 4 presents the data used to evaluate the approach, showing and commenting the obtained results, including a comparison with the state-of-the-art approaches; finally, Chapter 5 draws conclusions and suggests possible future works.

2 STATE OF THE ART

2.1 EXISTING METHODS

2.1.1 *First approach: dealing with control-flow structures*

The problem of extracting decision rules from a process model has not been treated vastly in the literature. The general idea, first proposed in [4] and then detailed in [5], is to consider every decision point as a classification problem; after the creation of a proper training set for each decision point, Decision Trees are used to solve the problem and extract the rules we are interested in.

The first step is to discover a Petri Net representing the process model starting from the Event Log, and this is usually done by applying existing Process Discovery algorithms. The first one that has been proposed is the *Alpha algorithm* [6]: it classifies the traces activities in four relations (direct succession, causality, parallel and choice) and then builds a Petri Net accordingly. The resulting model does not contain invisible activities, i.e., transitions that do not have a corresponding activity in the log; additionally, this algorithm is not able to deal with loops, it is not very effective against noisy data, and the obtained Petri Net might not be sound.

A better state-of-the-art alternative is the *Inductive algorithm* [7], which derives a directly-follows graph of the activities in the log, and then builds a Petri Net by detecting cuts in the graph. This algorithm is able to discover loops and invisible activities, extracting a sound model. This is indeed the most used algorithm to perform Process Discovery of a Petri Net starting from an Event Log.

Every time a process instance encounters a decision point in the Petri Net, i.e., a place with more than one outgoing arc, it means that a decision on which path to follow needs to be taken. Hence, we look at which transition has been executed afterwards, and we add a new example in the training set of that decision point. The new example will contain the attributes values from the Event Log before the choice has been made (i.e., at the decision point) and the transition that has been fired.

To identify which decision has been taken at each decision point, we have to look at the first activity observed immediately after the decision point: in a Petri Net, this corresponds to the transition pointed by the selected outgoing arc. Hence, a correct interpretation of the control-flow of the process model is needed and this may not be straightforward, especially in more complex

processes and nets. As noted in [5], three major issues can be identified when performing Decision Points Analysis:

- *Invisible activities*: transitions in the Petri Net which do not reflect actual activities recorded in the Event Log; they are only used for routing purposes inside the net, for example to skip other transitions.
- *Duplicate activities*: transitions in the Petri Net which correspond to the same activity in the Event Log. They have the same label in the net, and they cannot be distinguished by looking at the log.
- *Loops*.

To deal with invisible activities, the authors in [5] propose *Alg. 1* to track the next non-invisible transition in the net, using it as the selected activity after the decision point. The idea is to keep going through invisible transitions until an actual activity recorded in the Event Log is encountered. The search stops whenever an ambiguous place is found, for example a join place (i.e., a place with more than one incoming arc), therefore this approach may lose some information.

Duplicate activities are treated in a similar way, tracing the next visible unambiguous transition which allows to distinguish between the different paths, or until a join construct is encountered.

To handle loops, instead, they suggest to consider an activity as a possible target of a decision point according to its relative position, that is, before, inside or after the loop. Needless to say, loops combined with invisible or duplicate transitions are not easy to handle, at least not without information loss.

After analyzing all the traces in the Event Log, the obtained training sets (one for each encountered decision point) are fed to as many Decision Tree Classifiers in order to extract the rules governing the decisions. Each fitted model is able to discriminate between instances in the Event Log related to that decision point. The rule to enable transition t can be obtained by descending the Decision Tree from the root down to the leaf node with the corresponding label t, resulting in a conjunction of all the conditions on that branch. In case there are multiple leaves with the same target label t, then all these rules can be put in disjunction, since they can be seen as alternative conditions to execute the same transition t.

In conclusion, the obtained rule related to transition t is a disjunction of conjunctions. It explains, in terms of attributes values, the reason why certain process instances follow that specific path in the process.

The Decision Mining method proposed in [4] has been implemented in the ProM framework, which collects many plug-ins to perform Process Mining.

Algorithm 1 Recursive method proposed in [5] for specifying the possible decisions at a decision point in terms of sets of log events.

```
function determineDecisionClasses
    decisionClasses ← new empty set
    while outgoing edges left do
        currentClass ← new empty set
        t ← target transition of current outgoing edge
        if (t ≠ invisible activity) ∧ (t ≠ duplicate activity) then
            add l(t) to currentClass
        else
            currentClass ← traceDecisionClass(t)
        end if
        if currentClass ≠ ∅ then
            add currentClass to decisionClasses
        end if
    end while
    return decisionClasses
end function

function traceDecisionClass(t)
    decisionClass ← new empty set
    while successor places of passed transition left do
        p ← current successor place
        if p = join construct then
            return ∅
        else
            while successor transitions of p left do
                t ← current successor transition
                if (t ≠ invisible activity) ∧ (t ≠ duplicate activity) then
                    add l(t) to decisionClass
                else
                    result ← traceDecisionClass(t)
                    if result = ∅ then
                        return ∅
                    else
                        result ∪ decisionClass
                    end if
                end if
            end while
        end if
    end while
    return decisionClass
end function
```

2.1.2 *An alternative using alignments*

The approach proposed in [8] partially solves this problem using *optimal alignments*. A control-flow alignment between a trace in the Event Log and the Petri Net model is an ordered sequence of tuples: the first element in the tuple represents the move (i.e., the activity) in the log, while the second element represents the move in the model. If a transition in the model does not have a correspondence in the Event Log (for example, because it is an invisible transition), then the move in the log is depicted as $\gg$, and vice versa. Each alignment is characterized by a cost which quantifies how much the log trace deviates from the model: the optimal alignment between the trace and the net is the one with the lower cost.

The optimal alignment path is then used to characterize decision points, building their training sets. This method allows to select an entire path in the Petri Net model for each trace in the Event Log, therefore solving the problem of stopping on ambiguous places when looking for the next non-invisible transition. At the same time, only the optimal path from the Petri Net source to its sink is taken into consideration, leaving behind all the other possible paths involving invisible transitions. In conclusion, there is still information loss.

Even if the procedure to link Event Log data with proper decision points is different, the rules extraction part is unchanged with respect to the approach presented previously. A Decision Tree Classifier is fitted on each training set and rules are extracted following the conditions on their branches, as previously described.

2.2 BACKGROUND

2.2.1 *Event Log*

An Event Log contains the knowledge related to the executions of a business process: these data are usually stored automatically by an information system. However, this does not always happen: it is often the case that some information is not recorded because it is not considered relevant, because it is inferable from contextual information, or simply because the information system has not been implemented with the goal of exploiting the recorded data, for example by applying Process Mining techniques. This can be an issue when analyzing the process, since automatic methods are not able to deduce missing knowledge without the help of an expert.

More precisely, an Event Log is a multiset of sequences, where each sequence represents a specific process execution. Each sequence, also called *trace*, is an ordered collection of the events that occurred in that process instance, and it is uniquely identified by a *case id*. For each event in a trace,

the name of the executed activity is recorded, along with the related process attributes values, for example the timestamp, the resource that performed that activity, and other attributes depending on the particular process. An example of Event Log is shown in *Fig. 2.1*. A unique succession of events is a variant: therefore, all the traces composed by that course of activities can be grouped under that single variant.

From a technical perspective, the attributes in an Event Log can be distinguished between three major categories:

- *Continuous variables*: their values are numerical, and an order between numbers is defined. An example of continuous variable is a *Cost* attribute, which can take real positive values.
- *Categorical variables*: these variables are also known as *qualitative variables* and they take discrete values, either numeric or literals. An example of categorical variable is a *Resource* attribute, where each value corresponds to the name of the person which performed the corresponding activity. If numeric codes were used instead of real names, `Resource` would still be a categorical variable, since no reasonable ordering can be defined between the values.
- *Boolean variables*: attributes that can only take two possible values, `True` or `False`.

The size of the Event Log corresponds to the number of instances (i.e., rows) in it. Its dimensionality refers instead to the number of attributes present, also called features. Indeed, instances in the log can be represented in the feature space, in which every dimension corresponds to an attribute: an instance is a point in this space, and its coordinates are the values of the attributes.

An Event Log is seldom complete, but it contains missing values: these are usually represented using special characters and words, like `?`, `nan`, `NIL` and so on. Dealing with missing data is a common problem in the data mining and machine learning fields, and there exist different techniques able to take them into consideration.

2.2.2 *Process Mining*

Process Mining is a relatively new research area between data mining and process analysis, gathering together those techniques that are able to extract and model real processes, starting from Event Logs. As mentioned before, information systems are able to store the knowledge about business process executions, making massive quantities of data available and ready to be exploited. The analysis of these data can lead to a better understanding of the process, which in turn allows the process to be enriched and modified

Case ID	Timestamp	Activity	Resource	Cost
1	01-01-2018:08.00	Register (R)	Frank (F)	1000
2	01-01-2018:10.00	Register (R)	Frank (F)	1000
3	01-01-2018:12.10	Register (R)	Joey (J)	1000
3	01-01-2018:13.00	Verify-Documents (V)	Monica (M)	50
1	01-01-2018:13.55	Verify-Documents (V)	Paolo (P)	50
1	01-01-2018:14.57	Check-Vacancies (C)	Frank (F)	100
2	01-01-2018:15.20	Check-Vacancies (C)	Paolo (P)	100
4	01-01-2018:15.22	Register (R)	Joey (J)	1000
2	01-01-2018:16.00	Verify-Documents (V)	Frank (F)	50
2	01-01-2018:16.10	Decision (D)	Alex (A)	500
5	01-01-2018:16.30	Register (R)	Joey (J)	1000
4	01-01-2018:16.55	Check-Vacancies (C)	Monica (M)	100
1	01-01-2018:17.57	Decision (D)	Alex (A)	500
3	01-01-2018:18.20	Check-Vacancies (C)	Joey (J)	50
3	01-01-2018:19.00	Decision (D)	Alex (A)	500
4	01-01-2018:19.20	Verify-Documents (V)	Joey (J)	50
5	01-01-2018:20.00	Special-Case (S)	Katy (K)	800
5	01-01-2018:20.10	Decision (D)	Katy (K)	500
4	01-01-2018:20.55	Decision (D)	Alex (A)	500

Figure 2.1: Example of Event Log. All the events which are part of the same trace have the same *case id*.

to generate value. The Process Mining framework is depicted in *Fig.* 2.2, showing all the different components that take part in this field, along with their interactions.

The state of the art distinguishes between three types of Process Mining:

- *Process Discovery*, which allows to automatically discover a model of the process, based on the Event Log. Various algorithms have been developed to perform this task, like the Alpha algorithm or the Inductive algorithm already mentioned.
- *Conformance Checking*, which aims at assessing the compliance of the extracted model with the original Event Log. This is essential in order to review the soundness of the extracted model, and to understand if it complies with the information recorded in the log.
- *Process Enhancement*, which is used to extend an existing model with additional and useful information, using the Event Log as a source. As mentioned earlier, the log does not only contain a mere sequence of events, but also plenty of auxiliary knowledge which is not strictly related to the control-flow of the model.

Decision mining falls within the scope of Process Enhancement: given an Event Log and a process model, it aims at extracting the rules that drive the decisions inside the process. The process model can therefore be enriched with information that was not there before, but it was hidden inside the Event Log data.

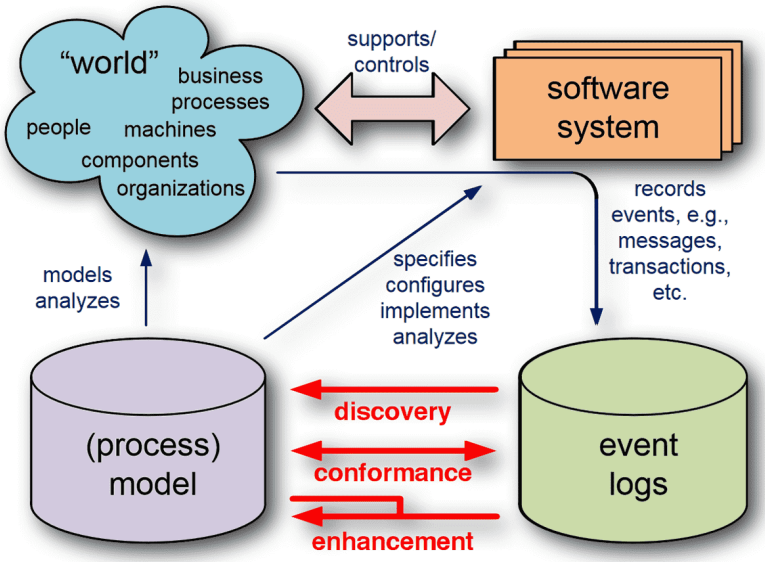


Figure 2.2: The Process Mining framework [2]. The red arrows shows the three types of Process Mining as defined in the Process Mining Manifesto.

2.2.3 *Petri Net*

A process can be represented through different notations, e.g., BPMN, Process Trees, Petri Nets, UML, EPC. Regardless of the chosen one, it must be able to correctly represent the model of a process, and it could be automatically extracted from an Event Log using Process Discovery techniques. We rely on Petri Nets, both because they are used also in state-of-the-art methods and because they are fitting to deal with Decision Mining.

A Petri Net, also known as *PT Net*, is a bipartite directed graph which has two types of nodes: places (P) and transitions (T). A place is represented by a white circle, and it represents a process state. A transition is represented by a rectangle containing the name of the activity. Places and transitions are connected through arcs. Places which are connected to a transition through output arcs are the transition input places, while transitions which are connected to a place through output arcs are the place input transitions.

Concurrency in a Petri Net is depicted through a split-join logic. An activity node is a split node when it has more than one outgoing arc; each arc is the beginning of a parallel branch, and every branch is executed concurrently. They are then joined together through a join node, which is an activity node with more than one incoming arc.

In the Process Mining context, Petri Nets could also include invisible transitions, depicted as black rectangles without any label: these are transitions that do not represent any actual activity recorded in the Event Log, and therefore they are not labeled in the net. They serve for routing purposes

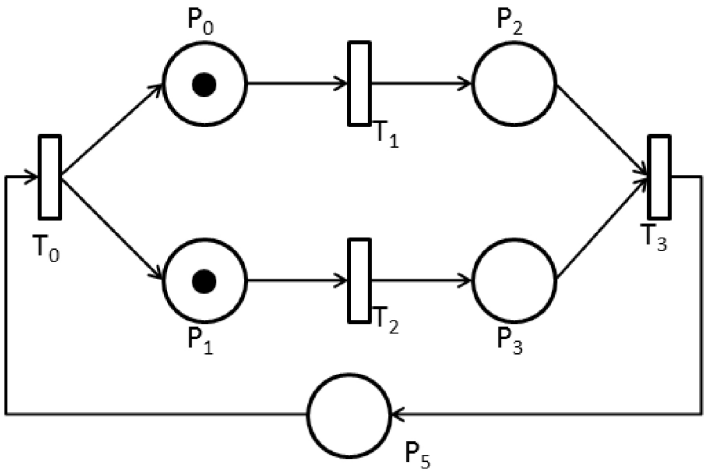


Figure 2.3: Example of Petri Net [9]. Places are represented by white circles, while transitions are depicted as white rectangles. Each black dot inside a place is a token.

only, for example to skip a certain activity or to split the process into parallel branches. Invisible transitions are completely transparent to the log, which instead stores only visible activities.

Every place can contain a certain number of tokens, and the distribution of tokens among all the places in the Petri Net is called marking. Each transition fires, i.e., is executed, only if every input place of the transition contains at least one token. After the firing of the transition, a token is removed from every input place and is added to each of the output places of the transition. Starting from the initial marking of the net, transitions fire one at a time until a final marking is reached or no transition can fire anymore. An example of a Petri Net is shown in *Fig. 2.3*.

When a Petri Net represents a business process, two special places are commonly present: a source and a sink. Process instances must start from the source place, then different transitions are fired, and finally process executions must arrive at the sink place. This is the stopping condition, which corresponds to a final marking in which only the sink place contains a token.

The formal definition of a Petri Net is the following:

Definition 2.1 (Petri Net)**.** A Petri Net is a tuple (P, T, F, M_0) where

- P and T are disjoint sets of places and transitions respectively.
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs.
- M_0 is the initial marking. A marking is a function associating a set of places with a natural number: $M : P \to \mathbf{N}$.

One could extend a Petri Net with additional capabilities, such as read and write operations. Read operations are also called *guards* and represent additional conditions on the firing of transitions: a guard is a boolean expression reading some (or all) of the process attributes and, if it evaluates to true, then the transition is allowed to fire. Write operations let transitions modify the values of specific process attributes upon firing.

A Petri Net enhanced with these extensions is called *Data Petri Net*, and its formal definition is the following:

Definition 2.2 (Data Petri Net)**.** A Data Petri Net is a tuple $(P, T, F, M_0, V, U, R, W, G)$ where

- (P, T, F, M_0) is a Petri Net.
- V is a set of variables.
- U is a function defining the admissible values for V.
- $R \in T \to 2^V$ is a read function providing each transition the set of variables it must read.
- $W \in T \to 2^V$ is a write function providing each transition the set of variables it must write.
- $G \in T \to G_V$ is a function providing each transition with a guard.

2.2.4 *Decision Mining*

Decision Mining is the sub-field of Process Mining which aims at enriching a given process model with rules. These rules characterize every decision point in the process model, telling which conditions a process instance must satisfy in order to follow that path. In the context of Petri Nets, Decision Mining is known as *Decision Points Analysis*: starting from a Petri Net and an Event Log related to the same process, the goal is to find the corresponding Data Petri Net. Indeed, the rules discovered through this approach are the guards of those transitions that immediately follow decision points. A decision point in a Petri Net is a place with at least two output arcs: it represents a state in which the process can take a different path depending on its attributes values.

This choice is usually mutually exclusive, hence only one path is taken among the alternatives; however, it may happen that the attributes values satisfy more than one guard, allowing the process instance to follow multiple paths at the same time. This may happen because the actual decision is taken by a human based on some context information which have not been included in the Event Log.

2.2.5 *Decision Tree Classifier*

Decision Tree Classifiers learning is a common supervised machine learning method for classification. This family of tasks aims at learning a function that maps inputs to outputs starting from a labeled training set. It consists of a set of training instances, each composed by some input features and a target label. The trained learner will then be able to output a label given an unseen instance as input.

An ideal optimal model would be able to predict the correct label for every possible instance given as input. This rarely happens in practice, since it would require a flawless training set covering all the possible cases. Therefore, the model is usually trained on a limited set of instances, which is the available data, the Event Log. The model only knows about those data, but at the same time it should be able to correctly classify unseen cases. Hence, the model should not overfit the training set, otherwise it would fail on new instances, despite being able to perfectly classify known ones.

The opposite also holds: we want to avoid underfitting, that is, a model which performs poorly on the training set. This happens when the model learns nothing actually useful from the training data, or just the dominant features such as the average value. In conclusion, the goal is to learn a model which is able to perform well both on known and unseen instances, avoiding overfitting as well as underfitting.

Decision Tree learners are used in different fields in order to obtain a predictive model by recursively partitioning a training set based on its features values. If the target variable takes discrete values, then the fitted model is known as Decision Tree Classifier, and it allows to classify the instances in the training set according to the values of their attributes.

The fitting process of a Decision Tree Classifier starts at the root node. A training set attribute is selected based on some goodness measure; if the attribute is discrete, then the training set is split in multiple sets, one for each possible attribute value; otherwise, a threshold is selected and the split is performed on that value. Each children node contains a split of the training set, and the procedure is repeated recursively until some conditions are met, or if no further splitting is possible. If a decision node cannot be split anymore, then it becomes a leaf node, and it is labeled with a target value from the target feature. An example of a fitted Decision Tree Classifier is shown in *Fig. 2.4*.

The selection of the split attribute and threshold (in case of continuous variables), as well as the termination conditions, is different according to the algorithm used.

One of the most commonly used algorithms for training a Decision Tree Classifier is the *C4.5 algorithm* proposed by John Ross Quinlan [10]. Each split attribute is the one which provides the highest information gain when

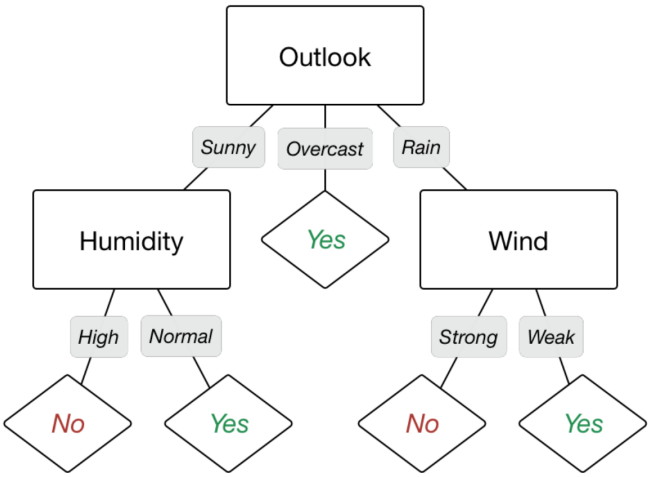


Figure 2.4: Example of Decision Tree. Every decision node is represented by a white rectangle containing the name of the attribute on which the split is performed. Each branch is labeled with the corresponding value for that attribute. Leaf nodes are represented by white diamonds containing the label of the predicted class.

splitting the (partitioned) set at the parent node. The recursion stops when all the instances in a set belong to the same target class, or when further splitting does not provide any information gain. Moreover, there could be additional rules to stop the splitting procedure: for example, when a node contains a minimum number of instances, or when a maximum tree depth has been reached. These conditions prevent the Decision Tree from overfitting, i.e., perfectly fitting the training set but failing to generalize on new, unseen data.

Definition 2.3 (Entropy)**.** Given a set of instances I with k different target values, the entropy of the set I is defined as:

$$\mathtt{Entropy}(I) = -\sum_{i=1}^{k} p_i \cdot \log_2 p_i$$

Where p_i is the fraction of instances with k as target value.

Definition 2.4 (Information Gain)**.** Given a set of instances I, a split attribute A with n different values, the information gain of the set I on the attribute A is computed as:

$$\mathtt{InformationGain}(I, A) = \mathtt{Entropy}(I) - \sum_{i=1}^{n} \frac{I[A = i]}{I} \cdot \mathtt{Entropy}(I[A = i])$$

Where $I[A = i]$ is the number of instances in I such that the value of attribute A is i.

The information gain is defined as the reduction of entropy resulting from splitting a set of instances according to an attribute (and a threshold, if the attribute is continuous). The entropy of a set is a measure of the skewness of its target values: if half of the instances in the set have as target value A while the other half have B, then the entropy of the set is exactly 1. The more unbalanced the set, the lower the entropy; a set with only one target value will have entropy 0. Seeking for the highest information gain is equivalent to looking for the split which minimizes the entropy in the resulting sets. The goal of the C4.5 algorithm is indeed to partition the instances based on their target value, so that each leaf node would contain instances with the same target value.

RULES EXTRACTION A fitted Decision Tree Classifier can be translated into a set of rules. Each branch of the Decision Tree composes a conjunction of atomic conditions, that are the splits on the path from the tree root to the leaf; then, branches which end up in leaves with the same label (i.e., target class, hence the name of the activity executed after the decision) are combined in disjunction. The final result is one rule for each target value, in general formed by disjunctions of conjunctions.

Alg. 2 shows how the rules extraction process is performed. First, conjunctive rules are gathered, one for each leaf node of the Decision Tree. Then, all the rules having the same target value are put in disjunction.

Algorithm 2 Method to extract rules given a Decision Tree.

```
function extractRules(decisionTree)
    rules ← ∅
    leafNodes ← getLeafNodes(decisionTree)
    for leaf ∈ leafNodes do
        branchRule ← getBranchRule(decisionTree, leaf)
        leafLabel ← getLeafLabel(leaf)
        rules ← addRule(rules, branchRule, leafLabel)
    end for
    seenTargets ← getSeenTargets(rule)
    for target ∈ seenTargets do
        rulesWithTarget ← getRulesWithTarget(rules, target)
        rulesWithTarget ← createDisjunctiveRule(rulesWithTarget)
        rules ← addRule(rules, rulesWithTarget, target)
    end for
    return rules
end function
```

RULES PRUNING Although the rules extraction process may be easy, the resulting rules may not. When the height of the Decision Tree is significant, the initial conjunctions of conditions may be quite long. Additionally, the presence of multiple leaves with the same target may result in a final rule composed by a considerable number of disjunctions. These two cases are not mutually exclusive: a deep Decision Tree with multiple leaves with the same target value would result in even longer rules.

The extracted rules perfectly represent the Decision Tree, but they may be difficult to read and understand. To solve this problem, Decision Trees and rules can be simplified to be shorter and easier to understand, but still being faithful to the original Decision Tree. Additionally, pruning helps with overfitting: a simpler model should be able to generalize better on unseen data.

Different methods have been proposed in the literature to prune Decision Trees or to directly simplify rules coming from them. In general, there are two possible approaches to prune Decision Trees: *pre-pruning* and *post-pruning*.

Pre-pruning, also called *stopping*, refers to those techniques which are able to stop the splitting procedure while fitting the tree. This is done by evaluating some quality measure, for example the information gain of a split, and by avoiding further splitting if this measure is below a certain threshold. These methods have the advantage of saving time, since the Decision Tree is pruned while it is being constructed. However, finding a good and general threshold for stopping is usually difficult, resulting in a simplification that is either too heavy or too shallow. Post-pruning techniques, on the other hand, work on an already fitted Decision Tree. They usually build new trees which are equivalent to the original one but more condensed.

The most common approaches are related to the latter family of pruning techniques: it is the case of *Cost-Complexity pruning*, *Reduced Error pruning* and *Pessimistic pruning* [11]. The first two methods need a separate test set, while the last one does not, and it is also much faster. Here, we present two state-of-the-art methods we implemented: the first one prunes the Decision Tree by replacing subtrees with leaves when possible, while the second one directly simplify the production rules that are extracted from the tree.

The pruning method proposed by J. R. Quinlan in its work about the C4.5 algorithm [10] is a post-pruning one which only requires the original training set. It is defined as a pessimistic pruning approach, and it works by recursively replacing subtrees with leaves according to the number of predicted errors. The algorithm is reported in *Alg. 3*.

Given the original Decision Tree, we consider every subtree having only leaf nodes as children. For each leaf, we call N the number of training instances that are contained in that leaf (i.e., the ones satisfying all the conditions along the branch from the tree root to that leaf). Among these, we call E the number

of training instances which are wrongly classified by that leaf (i.e., their target value is different from the label of the leaf).

Algorithm 3 Pessimistic pruning of a Decision Tree.

```
procedure pessimisticPruning(decisionTree, trainingSet)
    recurse = False
    bottomSubtrees ← getBottomSubtrees(decisionTree)
    for subtree ∈ bottomSubtrees do
        subtreeErrors = 0
        for leaf ∈ subtree do
            label ← getLabel(leaf)
            subtreeErrors += computeErrors(leaf, label, trainingSet)
        end for
        subtreeLabels ← getSubtreeLabels(subtree)
        for target ∈ subtreeTargets do
            errors[label] ← computeErrors(subtree, label, trainingSet)
        end for
        minErrorLabel ← getMinErrorLabel(errors)
        if minErrorLabel < subtreeErrors then
            subtree ← replaceSubtree(subtree, minErrorLabel)
            recurse = True
        end if
    end for
    if recurse = True then
        pessimisticPruning(decisionTree, trainingSet)
    end if
end procedure

function computeErrors(node, label, trainingSet)
    nodeInstances ← getNodeInstances(trainingSet)
    wrongInstances ← getWrongInstances(nodeInstances, label)
    upperBound ← getUpperBound(nodeInstances, wrongInstances)
    return nodeInstances · upperBound
end function
```

The error rate for that leaf is therefore *E/N*. The author's idea is to think of this as observing *E* events among *N* experiments: we can then compute the probability of error as the upper confidence bound of a binomial distribution having as parameter *N* and *E*, that is $U_{CF}(E, N)$ where the confidence level *CF* is suggested to be 25%. Finally, the total number of predicted errors for that leaf can be computed as $N \cdot U_{CF}(E, N)$ since *N* cases are covered by the leaf. The number of predicted errors for the entire subtree is simply the sum of these terms for all its leaves.

After computing the number of predicted errors for the original subtree, it is time to determine the same quantity for the subtree when replaced with a leaf, to understand if a substitution is needed. To do this, the subtree is substituted with a leaf having as target value each one of the labels of its leaves in turn. The computation is exactly the one described before.

At this point, the subtree is replaced with a leaf node having as target value the one which provided the minimum number of predicted errors, given that it is less than the number of predicted errors of the original subtree. Indeed, if the number of predicted errors of the original subtree is still lower, then we do not have to replace it, since the predictive performance would be worse.

After repeating the procedure for every subtree in the Decision Tree, the whole process is recursively applied on the pruned tree, until no more simplification is possible. The final result would be a pruned Decision Tree, with fewer nodes and lower height; hence, the rules extracted from it are possibly shorter, more readable and more easily understandable.

An alternative perspective for Decision Tree pruning has been proposed by J. R. Quinlan in [11] and it works directly on the rules that can be extracted from the original Decision Tree. These are also called *production rules*, since they are composed of a set of conditions which "produce", or lead to, a certain label. For every leaf, the approach extracts the corresponding production rule in the form $X_1 \wedge X_2 \wedge \cdots \wedge X_n$ then $\mathtt{class}\ c$ where $X_1, X_2, \ldots, X_n$ are all the conditions encountered along the path from the root to the leaf, and $\mathtt{class}\ c$ is the class of the leaf, i.e., its label.

Then, the left-hand side of the production rule is simplified by dropping some conditions, according to their relevance to the classification. This is done by isolating each X_i in turn, and considering the training examples that satisfy all the other conditions in the production rule; then, we build a 2×2 contingency table dividing these examples according to the satisfaction of X_i and the belonging to $\mathtt{class}\ c$. A Fisher's exact test is then performed on the contingency table to understand the relevance of X_i in the classification. The test returns a *p-value* which represents the probability that the division in the contingency table occurs from chance.

Having done this for every X_i in turn, the condition X_i which is least relevant for classification (i.e., the one with the largest p-value) is removed from the left-hand side of the production rule, provided that its p-value is greater than a predefined threshold. The Fisher's exact test is repeated recursively on the simplified production rule until no condition can be removed, or all conditions have been discarded.

This process, once applied to every production rule extracted from the Decision Tree (i.e., to every leaf) allows for an in-depth pruning of the original rules. Each simplified rule is a conjunction of conditions, and all the simplified rules having the same target class are put in disjunction, as explained before. The algorithm of this pruning technique is reported in *Alg. 4*.

Algorithm 4 Recursive simplification of production rules in a Decision Tree.

```
function simplifyProdRules(decisionTree, trainingSet, threshold)
    finalRules ← ∅
    for leaf ∈ decisionTree do
        prodRule ← getBranchRule(decisionTree, leaf)
        label ← getLabel(leaf)
        rule ← simplifyRule(prodRule, label, trainingSet, threshold)
        finalRules ← addRule(finalRules, rule, label)
    end for
    return finalRules
end function

function simplifyRule(prodRule, label, trainingSet, threshold)
    candidates ← ∅
    for rule ∈ prodRule do
        otherRules ← getOtherRules(rule, prodRules)
        table ← buildContTable(rule, otherRules, label, trainingSet)
        pValue ← fisherTest(table)
        if pValue > threshold then
            candidates ← addValues(candidates, (pValue, rule))
        end if
    end for
    if candidates ≠ ∅ then
        ruleRemove ← getRuleToRemove(candidates)
        productionRule ← removeRule(ruleRemove, prodRule)
        prodRule ← simplifyRule(prodRule, label, trainingSet)
    end if
    return prodRule
end function
```

2.3 RELATED WORKS

2.3.1 *Overlapping Rules*

The rules extraction process leads to the discovery of mutually-exclusive rules: every branch of the Decision Tree represents a conjunctive condition which allows the execution of the transition specified by the corresponding leaf. This is intrinsic to the nature of Decision Trees, since a branch leads to exactly one leaf.

Although this is perfectly acceptable for most of the business processes, others may not be fully deterministic, or some information may not have been recorded in the Event Log. In these cases, rules that allow a process instance to follow multiple choices at the same time may fit the data better, even though the precision is lower. A technique to deal with such situations, and to obtain the so-called *overlapping rules*, has been developed in [12].

Fitness and precision are two distinct concepts that allows to establish the quality of the extracted guards for output transitions of a decision point. The authors in [12] define them for every decision point as follows.

Definition 2.5 (Place fitness)**.** Given E_p the subset of the Event Log containing only those events which are related to output transitions of place p, the place fitness of p is defined as:

$$\mathrm{Fitness}(E, p) = 1 - \frac{|\, e \in E_p \,|\, \text{Guard of } trans(e) \text{ is violated} \,|}{|\, E_p \,|}$$

where $trans(e)$ returns the Petri Net transition corresponding to the event e in the log. Therefore, the fitness of place p decreases the more output transitions of place p are executed (i.e., are observed in the Event Log) violating the guards.

Definition 2.6 (Place precision)**.** Given E_p the subset of the Event Log containing only those events which are related to output transitions of place p, the place precision of p is defined as:

$$\mathrm{Precision}(E, p) = \frac{\sum_{e \in E_p} |\, obs_p(e) \,|}{\sum_{e \in E_p} |\, pos_p(e) \,|}$$

where $pos_p(e) : E \to \mathbb{N}$ returns the number of possible executions of output transitions of place p (according to the guards) before event e occurred, while $obs_p(e) : E \to \mathbb{N}$ returns the number of actually observed executions. This means that the place precision of place p decreases the more the guards of its output transitions are overlapping.

The starting point is a set of observation instances for every decision point: each instance is composed of the set of attributes values observed before the

decision is taken, and the target value, i.e., the following transition. These are the datasets mentioned before, and they can be obtained through different approaches, such as the ones described in [4] or [8], but also the one proposed in this thesis work. Subsequently, a Decision Tree Classifier is fitted on each of these datasets, exactly as described above.

Afterwards, each Decision Tree is considered in order to possibly extend the extracted rules with overlapping ones. More precisely, every leaf node l is analyzed and taken into account only if it contains at least one misclassified instance (i.e., not all the instances that follow that branch have as target value t the one labeling the leaf). For every leaf node l with misclassified instances, a new Decision Tree is fitted on those instances. Indeed, the idea is to focus on those instances in order to further discriminate between them and possibly find rules that overlap with the existing ones.

The new Decision Tree can contain either only one leaf (i.e., no suitable split is found) or more leaves. In the first case, the label of the single leaf would be the target value t' which appears more often among the training instances of this new tree. The original rule for this target value t' is therefore expanded adding in disjunction the conjunctive conditions *expr* that led to leaf l in the original Decision Tree. Indeed, these misclassified samples would follow transition t' in the Petri Net model.

Concerning this first case, the original rule for target t' is expanded as follows: $\mathtt{rules}(\mathtt{t}') \leftarrow \mathtt{rules}(\mathtt{t}') \vee \mathtt{expr}$.

In the second case, for every leaf l' of the new Decision Tree, initially we have to put in conjunction the conditions *expr* that led to leaf l of the original Decision Tree and the ones *subExpr* that lead to leaf l'. This is because, in order to reach those wrongly classified instances, we have to descend the original Decision Tree down to leaf l and then to further descend the new Decision Tree down to leaf l'. As always, the resulting rules for all the leaves l' with the same target value t' are put in disjunction between them. Finally, the original rule for each target value t' is expanded adding in disjunction the obtained disjunction of conjunctions.

Concerning this second case, the original rule for target t' is expanded as follows: $\mathtt{rules}(\mathtt{t}') \leftarrow \mathtt{rules}(\mathtt{t}') \vee \bigvee_{(\mathtt{subExpr},\mathtt{t}') \in \mathtt{subTree}_l} (\mathtt{expr} \wedge \mathtt{subExpr})$.

Alg. 5 shows the procedure to discover overlapping rules.

2.3.2 *Discovering Conjunctive Conditions with Daikon*

An alternative approach to perform Decision Points Analysis given an Event Log and a process model (e.g., a Petri Net) relies on the *Daikon* invariants detector and has been presented in [13]. This method does not make use of Decision Trees.

Algorithm 5 Method to discover overlapping rules.

```
function discoverOverlappingRules(decisionTree, trainingSet, rules)
    leavesWithWrong ← getLeavesWithWrong(decisionTree)
    for leaf ∈ leavesWrongData do
        subRules ← ∅
        expr ← getBranchRule(decisionTree, leaf)
        wrongInstances ← getWrongInstances(leaf)
        subTree ← fitDecisionTree(wrongInstances)
        subTreeNodes ← getNumSubTreeNodes(subTree)
        if subTreeNodes > 1 then
            for subLeaf ∈ subTree do
                subExpr ← getBranchRule(subTree, subLeaf)
                subExpr ← combineExprs(expr, subExpr)
                subLabel ← getLeafLabel(subLeaf)
                subRules ← addRule(subRules, subExpr, subLabel)
            end for
            subRules ← disjSameTarget(subRules)
            for target ∈ subRules do
                subRuleTarget ← getRulesTarget(subRules, target)
                rules ← addRule(rules, subRuleTarget, target)
            end for
        else
            target ← getMostPredictedTarget(subTree)
            rules ← addRule(rules, expr, target)
        end if
    end for
    return rules
end function
```

Daikon is an implementation of dynamic detection of likely invariants, that is, it reports likely program invariants. An invariant is a property that holds at a certain point in a program, e.g., $x.field > abs(y)$, $y = 2^x + 3$, *array* a *is sorted* and so on. Although Daikon was born with the aim of finding likely invariants starting from a program source code, it also supports comma-separated-values (csv) files. The spreadsheet data is first processed by a *Perl* script in order to obtain correctly formatted files to be used by Daikon, which then extracts the invariants. This is what is needed in the Decision Points Analysis field, since there is not a program source code, but instead datasets of observation instances. The approach suggested in [13] can only be used on binary decision points; however, n-ary decision points can be easily converted into a sequence of multiple binary decision points.

First, we split the set of observation instances related to a decision point into two sets, according to the target value. Therefore, each one of these sets has a single target value, and it contains the observation instances related to that specific branch of the decision point. Then, we use Daikon on each of these two sets: it returns a list of invariants which are valid on that set. These can be seen as properties of that set: for example, if the attribute *skip_everything* always takes the value *True* in that set, we will get the invariant *skip_everything == True*.

The discovered invariants are then used to build one conjunctive expression for each branch. Given the list of invariants for a set reported by Daikon, we compute the information gain for each invariant, and we start building the resulting conjunctive expression P by picking the one with the highest information gain. Then, we iteratively add a new invariant q to the conjunctive expression only if the conjunction $P \wedge q$ increases the information gain with respect to P. When all the invariants have been analyzed, we return the resulting conjunctive expression. Note that this is done also for the other list of invariants (the one related to the other branch of the decision point).

Finally, since we are considering binary decision point, the idea proposed in [13] is to have a mutually exclusive choice between them. Therefore, one conjunctive expression must be set as the negation of the other one. To do this, considering C_1 and C_2 as the resulting expression, we can distinguish between four cases:

- If C_1 is not empty but C_2 is, we put $C_2 = \neg C_1$.
- If C_1 is empty but C_2 is not, we put $C_1 = \neg C_2$.
- If C_1 and C_2 are not empty, we compute the information gain for each of them, we keep the one with the highest information gain and we put the second one as the negation of the first one.
- If C_1 and C_2 are both empty, simply return *None* for both.

In order to discover more interesting invariants related to continuous variables, a massive help comes from enhancing the initial dataset with so-called *latent variables*, as suggested in [13]. Each continuous variable is combined with every other continuous variable using the four basic operators $[+, -, *, /]$. Note that this could be easily extended adding other operators. The drawback is that adding all these combinations to the dataset vastly increases its dimensionality, resulting in longer execution times.

3

METHODS

3.1 RUNNING EXAMPLE

In order to support the explanation of the method proposed in this thesis, we introduce a synthetic Data Petri Net depicting an example business process. The model is shown in *Fig. 3.1*. Invisible transitions are labeled for ease of use, so that we can conveniently refer to them. Every logical expression corresponds to a guard, and it is linked to the related transition through a dotted line.

The model represents an exemplified loan request process, and does not aim at simulating a real one. Every trace execution begins at the *start* place on the left, and terminates at the *sink* place on the right. The first performed activity is always the *Request* of the loan; then, the customer needs to *Register* a new account, if they do not have one already.

Afterwards, the model splits in two parallel branches. The first one performs a *Check* of the received loan request, only if the desired amount is higher than a predefined threshold or if the loan request has been rejected earlier on. The second one draws up a document (*PrepDoc*) unless it is already up to date. Additionally, these two parallel activities are skipped if the loan request comes from a premium customer (*skip_everything == True*).

Anyhow, a subsequent *FinalCheck* is performed to decide the fate of the loan request. Indeed, it can be accepted and therefore going through the *OK* transition, or it can be rejected hence ending up in a *NOK* transition. Both paths lead to the termination of the process. A third possibility imposes a *recheck* of the loan request: in this case, the process must go through a *Check* no matter what.

The running example model allows to take into consideration peculiar control-flow structures in the Petri Net, such as invisible transitions, parallel branches and loops. In particular, the loop is composed by activities *Check, PrepDoc, FinalCheck*. This is practical for developing new methods for Decision Points Analysis, since we can cover more complex cases.

To create a synthetic Event Log for this process, we choose random values for the attributes at the beginning of every trace. The sequence of activities in the trace is generated by arbitrarily firing transitions in the model until the final marking (i.e., the sink) is reached. The shortest variant we can have is $\langle$*Request, FinalCheck, OK*$\rangle$ or $\langle$*Request, FinalCheck, NOK*$\rangle$. Assuming that the loop is executed, we get instead $\langle$*Request, FinalCheck, Check, FinalCheck, OK*$\rangle$ and similarly with *NOK*.

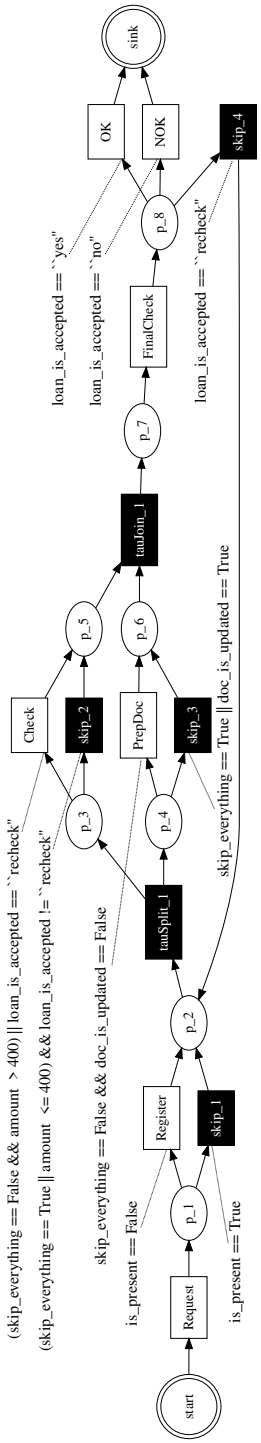


Figure 3.1: Data Petri Net of the running example. Every process instance begins at the *start* place on the left and terminates at the *sink* place on the right. Places are represented by ovals, while transitions are depicted by rectangles: white for visible activities and black for invisible ones. Logical expressions correspond to the guards, and they are linked to the related transitions through dotted lines.

3.2 PROPOSED METHOD

As mentioned in the previous chapter, the approaches presented in both [4] and [8] are prone to information loss: the former stops tracking invisible transitions when encountering ambiguous places, the latter relies on the optimal alignments between the Event Log and the Petri Net model, without considering all the other possible paths. This information loss results in less complete datasets for the decision points, hence in less accurate Decision Trees and, finally, in less precise rules.

The method proposed in this thesis allows to take into consideration more than one path among the ones between every pair of activities observed in the Event Log. As mentioned before, the Inductive algorithm discovers a Petri Net model representing a business process which usually contains invisible activities. Therefore, given two visible transitions in the model, there could be more than one viable path to reach the second one from the first one, traversing invisible activities. The goal is to reduce information loss, better exploiting the data contained in the log.

To achieve this result, first the Event Log is processed in order to create one dataset for each decision point in the Petri Net model. This is done by scanning the Event Log variants, and storing the decision points involved in all the possible paths between every pair of activities; this knowledge is then exploited to create the datasets, using the attributes values in the Event Log. Finally, a Decision Tree learner is fitted on each one of these datasets and it is then used to infer the transitions guards.

3.2.1 *Decision Points extraction*

In order to properly create a dataset for each decision point, we first need to store which decision points are involved between an activity A in the Event Log and the next one B. This way, for each decision point found, a new instance will be added to the corresponding dataset containing the attributes values in the Event Log up to activity A and the related target transitions (i.e., the transitions followed at every decision point).

Given the presence of invisible transitions, there can be multiple paths between transition A and transition B in the Petri Net model of the process. This also means that, among the decision points found in the paths, all of them have an invisible transition as target, except for the last one, which has transition B. In order to discover the decision points encountered along all the possible paths between a pair of activities, we developed a depth-first search algorithm.

FIRST VERSION Scanning every event in a variant in the Event Log, the algorithm starts from the transition in the Petri Net corresponding to the

current event B and performs a backward depth-first visit of the net, passing through invisible transitions until it reaches the previous activity A observed in the log. Therefore, this search is executed once for every event in the variant except for the first one. The backward search stops whenever the previous activity in the log has been found, or if there are no more invisible transitions to follow. In case the previous activity in the log is encountered, then the algorithm adds all the decision points along the followed path and the related targets (i.e., the decisions that have been taken) to a data structure.

More specifically, the structure of the algorithm is recursive: each level of recursion focuses on a transition in the model and it extracts all the backward activities, that are the transitions placed one step before the current one. Indeed, given the bipartite nature of the Petri Net, places are input nodes to transitions, and transitions are input nodes to places. Hence, given a transition, we look at its input places, and for each of them, we collect its input transitions: these are the backward activities mentioned before.

The algorithm starts by following one of the input arcs of the current transition: this leads to one of its input places. Then, it looks at the input transitions of that place: if the previous activity in the log is among them, then it stops following that specific path, adding the place as a decision point (if it is one) along with the decision that has been taken, which is the current transition. The algorithm still explores the paths related to the other input places, since they could also lead to the previous activity in the log.

Otherwise, if the previous activity in the log is not among the input transitions of the place, then the algorithm performs a recursion, following every invisible transition between them. If all these transitions are visible, then the algorithm simply stops the search along this path, since it cannot continue.

Every time a level of recursion returns forward along the followed path, it also signals if the previous activity in the log has been found, either by the current level of recursion or by lower levels. If so, then the higher level of recursion adds the current input place as a decision point. As mentioned earlier, every time a decision point is added to the data structure, this information is also linked to the decision that has been taken, which corresponds to the transition the algorithm finds itself at. Indeed, the path followed by the backward search traversed that transition, which becomes the decision taken at the current input place.

When the algorithm terminates its execution, the data structure containing the decision points and related targets is added to a more global mapping, which links this collection to the corresponding variant and current activity. This mapping is then exploited by the second part of the proposed method, when constructing the decision points datasets.

As highlighted in the previous chapter, a Petri Net model of a business process can often contain loops. By performing a depth-first search in the

net, this is an issue that needs to be addressed. Since the algorithm follows invisible activities backward, it may happen that the algorithm loops endlessly, following a cycle composed of invisible transitions.

To avoid this problem, just before the algorithm performs a recursion towards an invisible transition, it adds that transition to a list, passing it to the lower level of recursion, and it removes it just after the lower level returns. This way, the algorithm actually performs the recursion towards an invisible transition only if that transition has not been visited by that specific line of recursion.

Otherwise, it means that the path realizes a loop composed of invisible transitions and can be included by the algorithm, since it corresponds to a loop completely inside the path between the pair of activities analyzed. The discovered loop does not contain any visible activity, and therefore it is transparent to the Event Log. The Petri Net model allows it, hence we add the visited decision points to the data structure.

Keeping the decision points belonging to a loop of invisible activities is correct only assuming that the pair of activities analyzed by the algorithm is reachable. An activity is reachable if the backward depth-first search is able to get to it starting from the current activity. This is not always true, for example when considering two parallel activities, or when the extracted Petri Net model does not represent the Event Log with perfect accuracy.

To solve this problem, the algorithm should not always consider the previous activity in the log as the one to be reached by the backward search. Instead, it should choose the most recent activity which can be actually reached, starting from the current activity. We implemented this method manually, at least initially, encoding a list of reachable activities starting from every other activity.

Finally, we also implemented a slightly different version of the algorithm in order to detect decision points between the sink of the Petri Net and the last observed activity in the Event Log. Indeed, it may happen that the last executed activity is not immediately before the sink, but there could be other transitions in between, according to the model. These activities have not been recorded in the log, therefore they have not been executed: this means that, from the last observed activity, the process execution traverses a path composed of invisible transitions until it reaches the sink of the net, skipping visible transitions.

The general structure of the algorithm is similar to the one explained before. The main difference is that now the method follows every invisible transition backward until it finds a decision point that has been already stored during the sliding of the current variant. In that case, it returns adding all the decision points present along the traversed paths.

The algorithm implementing the backward depth-first search is reported in *Alg. 6*, while the method selecting the most recent reachable activity is reported in *Alg. 7*.

EXAMPLE To clarify better the behavior of the proposed method, we illustrate an example of execution of the backward depth-first search. To support the explanation, the Data Petri Net of the running example is reported in *Fig. 3.2*, with arcs colored in different ways according to the paths explorations.

Consider the sequence $\langle Register, FinalCheck \rangle$ from the running example. The method starts at the *FinalCheck* transition and follows its only input arc, finding place *p_7*. The only backward transition is *tauJoin_1* and it is an invisible one, hence the algorithm performs a recursion.

The new level of recursion finds itself at transition *tauJoin_1* and can follow two input arcs. It starts by following one of them, for example the one towards place *p_5*. Among the input transitions of *p_5*, the activity we are looking for, i.e., *Register*, is not present. Therefore, the algorithm goes on with the recursion, through the only invisible transition *skip_2*.

The recursion goes on through *tauSplit_1*, where the algorithm can follow two paths backward: the one towards *skip_1* and the one towards *skip_4*. The recursion on the latter one does not lead to anything useful, since it encounters activity *FinalCheck* and no invisible transitions to keep going. The recursion on the former, instead, brings the algorithm to transition *skip_1* where finally activity *Register* is found, since it is one (the only one, in this case) of the backward activities with respect to *skip_1*.

At this point, the algorithm signals that the wanted activity has been found and returns to the upper level of recursion. This means that the traversed path is followed in the opposite direction, adding decision points and related targets that are found on the way. In this case, the method would add $\langle p\_1, skip\_1 \rangle$ and $\langle p\_3, skip\_2 \rangle$.

Arriving again at transition *tauJoin_1*, the algorithm has yet to follow the other unexplored path, which is the one towards place *p_6*. The routing is similar to the one explained before, until it arrives to transition *skip_1*. Once more, the method signals that the target has been found and starts retracing back its steps adding decision points, in this case $\langle p\_1, skip\_1 \rangle$ and $\langle p\_4, skip\_3 \rangle$.

Finally, having explored all the input arcs of transition *tauJoin_1*, the algorithm can keep returning to the upper level of recursion towards place *p_7* and finally transition *FinalCheck*. In conclusion, the decision points and related targets for the current variant and current activity (the second one of the variant, i.e., *FinalCheck*) are the following: $\langle p\_1, skip\_1 \rangle$, $\langle p\_3, skip\_2 \rangle$ and $\langle p\_4, skip\_3 \rangle$.

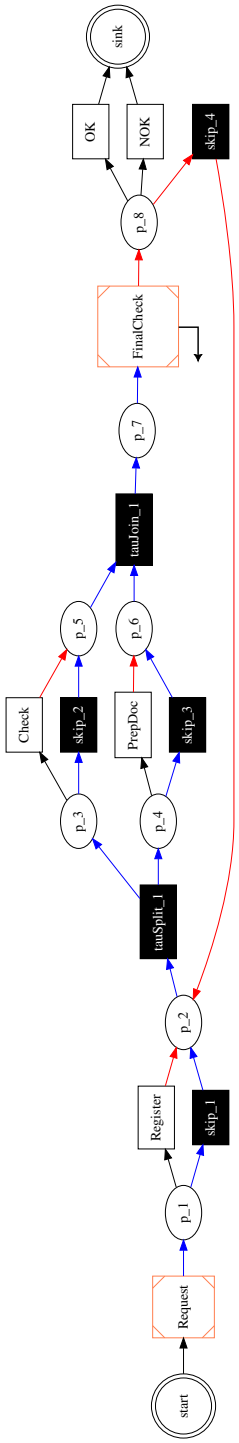


Figure 3.2: Example of an execution of the proposed method. The algorithm starts from transition *FinalCheck* and follows the net backward until it finds transition *Register*. Arcs highlighted in blue show the paths that are actually selected by the method: decision points on these paths are stored. Arcs highlighted in red represent instead paths that are explored by the algorithm but that are not chosen in the end.

Algorithm 6 Recursive method to extract decision points and related targets by performing a backward depth-first search of the Petri Net through invisible transitions starting from currAct up to prevAct.

```
function BWDDFS(prevAct, currAct, passedActs)
    prevActFound = False
    for inputArc ∈ getInputArcs(currAct) do
        inPlace ← getSource(inputArc)
        backTrans ← ∅
        for innInputArc ∈ getInputArcs(inPlace) do
            transition ← getSource(innInputArc)
            backTrans ← addTrans(transition, backTrans)
        end for
        if prevAct ∈ backTrans then
            prevActFound = True
            if isDP(inPlace) then
                mapDPs ← updateDPs(inPlace, currAct, mapDPs)
            end if
            continue
        end if
        invActs ← getInvisibleTransitions(backTrans)
        for invAct ∈ invActs do
            if invAct ∉ passedActs then
                passedArcs ← addArc(invAct, passedArcs)
                mapDPs, found ← BWDDFS(prevAct, invAct, passedActs)
                passedArcs ← removeArc(invAct, passedActs)
                if found == True ∧ isDP(inPlace) then
                    mapDPs ← updateDPs(inPlace, invAct, mapDPs)
                end if
                prevActFound = prevActFound ∨ found
            else
                prevActFound = True
                if isDP(inPlace) then
                    mapDPs ← updateDPs(inPlace, currAct, mapDPs)
                end if
            end if
        end for
    end for
    return mapDPs, prevActFound
end function
```

Algorithm 7 Method to select the most recent activity in the Event Log that can be reached by the current one. It also runs the backward depth-first search of the chosen activity, if it exists.

```
function extractDPs(prevSequence, currTrans)
    mapDPs ← ∅
    prevSequenceReversed ← reverseSequence(prevSequence)
    for prevAct ∈ prevSequenceReversed do
        if areActsReachable(prevAct, currTrans) then
            mapDPs, found ← bwdDFS(prevAct, currTrans, ∅)
            break
        end if
    end for
    return mapDPs
end function
```

PROBLEMS Although this method is able to explore all the paths between a pair of transitions, collecting decision points and related targets, it is not perfect. The biggest drawback is that it is not able to detect decision points in particular situations involving parallel branches. More precisely, if the activity to be reached by the backward search is located in one of possibly many parallel branches in the Petri Net, and some other branch is not visited according to the Event Log, this means that it is traversed through invisible transitions only. In that case, the algorithm should store the decision points inside that unexplored branch, having invisible activities as targets. However, the proposed method is not able to do this: once the previous activity in the log is reached, the other parallel branches do not contribute to the stored decision points.

For example, consider the sequence ⟨*Request, Register, Check, FinalCheck*⟩. The algorithm starts at transition *FinalCheck* and performs a backward search until it finds transition *Check*. It does that correctly, without adding any decision points since *p_5* and *p_7*, the only places on the selected path, are not actual decision points.

However, the variant does not contain activity *PrepDoc*. This is a parallel activity with respect to *Check* and therefore also the branch *tauSplit_1*, *p_4*, *skip_3*, *p_6*, *tauJoin_1* should have been accepted, adding ⟨*p_4, skip_3*⟩ as useful decision point. The algorithm is not able to discover it, since that path does not lead to *Check*.

Another issue with this approach is the manual encoding of reachable transitions. Indeed, if the currently analyzed activity cannot reach the previous one in the Event Log, we go back in the log until we find a reachable activity. To understand if an activity is reachable in the Petri Net, the algorithm relies on a manually-written list containing the reachable transitions starting from

every other one. Therefore, there is the need for an automatic method able to deal with this problem.

Aiming at finding a solution to these pitfalls, we developed a new version of the algorithm. This time, the backward search not only takes in consideration the previous activity in the log, but the entire sequence of activities encountered in the variant so far. Going backward, the behavior when the previous activity in the log is found is unchanged; instead, every time the algorithm reaches any other activity already seen in the variant, then it decides whether adding the decision point or not according to some conditions. These conditions are based on decision points already seen in the variant and on the presence of loops, e.g., decision points are taken into consideration if the process instance is performing a loop, even if they have already been observed in the variant, since they can be encountered again.

In other words, this new method traces back to every previous activity it finds, storing the decision points in all those paths between the current activity and those ones. This allows to solve the problem mentioned before.

Considering the troublesome example presented earlier on, and therefore the sequence $\langle$*Request, Register, Check, FinalCheck*$\rangle$, the algorithm now chooses two paths: one between *Check* and *FinalCheck* and the other between *Register* and *FinalCheck*, passing through transitions *tauSplit_1*, *skip_3* and *tauJoin_1*. In conclusion, $\langle$*p_4, skip_3*$\rangle$ is stored as useful decision point. Note that, in principle, the newly discovered path would also add *p_2* as decision point, if it were one. However, the conditions inserted in the modified method prevent this, since *p_2* would have been already stored during the current variant when considering the pair of activities *Register* and *Check*.

This approach also solves the drawback of manually encoding reachable transitions, since it considers the whole sequence of activities seen in the variant so far and it traces back in the net until it finds the latest ones.

However, it is not capable of finding decision points correctly in more complex situations, not because it misses some possible paths, but because it considers additional, erroneous ones. Indeed, given certain variants and nets, this new version of the algorithm may find additional decision points located on paths already explored previously during the variant, since the newly introduced conditions are not able to cover all possible situations. In conclusion, the method is not fully generalizable since it may find incorrect decision points in particular occurrences.

In our opinion, it seems clearly better to prefer decision points on paths that could have happened during the process execution, rather than on ones that for sure never happened. Indeed, the alternative paths composed of invisible transitions that our method finds, actually represent routes that the process could have followed during its execution. They are not necessarily followed, since they are introduced in the Petri Net model with the only goal to cover all the possible cases in the Event Log.

Therefore, we realized that exploiting the modified algorithm could be precarious, since in particular conditions it may store decision points which are incorrect for the current event. Instead, the original version is able to discover the right decision points every time, losing some secondary path in infrequent situations. Still, in general it is able to discover much more with respect to state-of-the-art methods.

SECOND VERSION Since adapting the new version to every possible situation is definitely not straightforward, we reconsidered the original method. We modified it in order to avoid manually encoding all reachable activities, so that the method can be exploited on any Petri Net model.

More precisely, instead of checking if two activities are reachable looking at the manually-written list, the backward depth-search algorithm looks for the previous activity in the log and, if not found, the search is repeated looking for the activity prior to that, and so on. In case the current activity cannot reach any of the previous activities, then no decision points are stored. This possibly repeated search can be computationally expensive, especially when dealing with bigger Petri Nets with many invisible transitions; however, it has the advantage of correctly dealing with non-reachable activities without the need of manually encoding information.

It is not possible to reach an activity from a previous one in the variant in two distinct situations. The first one is when the two activities are parallel, i.e., they are located in two different parallel branches in the Petri Net. The second one is when the model does not perfectly represent all the process executions in the Event Log: in this case, the two activities can be actually executed one after the other, but the model is not able to represent that particular sequence.

In our modified approach, we condense these two cases into a single one: if the two activities are not reachable, the algorithm is run again looking for an even previous activity, as explained before. If the two initial activities are not reachable because the model does not represent faithfully the log, then this approach has the effect of possibly storing decision points already encountered in the variant. This is not a major issue: the datasets related to those decision points would contain additional data linked to a process instance that traversed them anyway.

Another issue is related to loops. In the original method, whenever the backward depth-first search algorithm encountered an invisible transition already seen along the path, it meant that a loop composed of invisible transitions was possible. However, this was true assuming that the two involved activities were reachable.

Having modified the algorithm, this does not always hold. Therefore, we ignore this kind of paths, which are not necessarily followed by the process, being composed of invisible activities only. In conclusion, loops of invisible transitions completely inside the path between the two involved activities

are now disregarded. The only kind of loop composed of invisible activities that is still considered is the one starting and terminating at the activity that needs to be reached, since the algorithm enters the base case (i.e., the target activity is reached).

The modified version of the algorithm implementing the backward depth-first search is reported in *Alg. 8*, while the modified version of the method selecting the previous activity to be searched is reported in *Alg. 9*.

Algorithm 8 Modified version of the recursive method to extract decision points and related targets.

```
function BWDDFS(prevAct, currAct, passedActs)
    prevActFound = False
    for inputArc ∈ getInputArcs(currAct) do
        inPlace ← getSource(inputArc)
        backTrans ← ∅
        for innInputArc ∈ getInputArcs(inPlace) do
            transition ← getSource(innInputArc)
            backTrans ← addTrans(transition, backTrans)
        end for
        if prevAct ∈ backTrans then
            prevActFound = True
            if isDP(inPlace) then
                mapDPs ← updateDPs(inPlace, currAct, mapDPs)
            end if
            continue
        end if
        invActs ← getInvisibleTransitions(backTrans)
        for invAct ∈ invActs do
            if invAct ∉ passedActs then
                passedArcs ← addArc(invAct, passedArcs)
                mapDPs, found ← BWDDFS(prevAct, invAct, passedActs)
                passedArcs ← removeArc(invAct, passedActs)
                if found == True ∧ isDP(inPlace) then
                    mapDPs ← updateDPs(inPlace, invAct, mapDPs)
                end if
                prevActFound = prevActFound ∨ found
            end if
        end for
    end for
    return mapDPs, prevActFound
end function
```

Algorithm 9 Modified version of the method running the backward search on the previous activities until a reachable one is found.

```
function extractDPs(prevSequence, currTrans)
    mapDPs ← ∅
    prevSequenceReversed ← reverseSequence(prevSequence)
    for prevAct ∈ prevSequenceReversed do
        mapDPs, found ← bwdDFS(prevAct, currTrans, ∅)
        if found == True then
            break
        end if
    end for
    return mapDPs
end function
```

3.2.2 *Datasets creation*

Having stored the valid decision points for every event in every variant in the Event Log, we can now use this information and the attributes values in the log to create a dataset for each decision point. These datasets will then be used as training sets to learn Decision Tree Classifiers.

For every event except for the first one of every trace, the algorithm retrieves the valid decision points and their targets (the ones extracted beforehand) related to the considered activity in the corresponding variant. These decision points are the ones encountered between the current activity and the previous reachable one. For every one of these decision points, the algorithm adds to the corresponding dataset a new instance, composed by the attributes values from the Event Log and the related target value extracted in the depth-first search. If an attribute is not present in the dataset, then the algorithm adds it as a new column, possibly filling the previous entries with a missing value indicator.

Note that the attributes values added to the datasets are the ones in the trace up to the previous event. This is because the valid decision points are those encountered during the backward depth-first search between the current event and the previous one. Therefore, the attributes values related to the analyzed event have not been observed yet, since the decision has not been taken yet.

For this reason, before analyzing the next event in the variant, the algorithm stores the current attributes values, retrieving the ones related to the considered event in the log. More precisely, the attributes values stored by the algorithm are always the most recent ones, according to the order as they appear in the trace. For example, if an attribute has its value set at the first event of the trace, and it is not present anymore in the sequence, then at the

skip_everything	amount	loan_accepted	...	target
True	800	?	...	*skip_2*
True	800	recheck	...	*Check*

Table 3.1: Partial dataset of decision point *p_3* of the running example

currently evaluated event the algorithm keeps that observed value for that attribute.

Consider the variant *Request, FinalCheck, Check, FinalCheck, OK*. The dataset related to decision point *p_3* will be filled with two instances, since the variant crossed it twice: the first time going from *Request* to *FinalCheck* through *skip_2*, while the second time going from *FinalCheck* to *Check*.

For every trace in the aforementioned variant, the procedure scans the activities in it, selecting the proper decision points extracted during the first step. In this case, when reaching the first *FinalCheck* activity, the valid decision points are of course $\langle p\_1, skip\_1 \rangle$, $\langle p\_3, skip\_2 \rangle$ and $\langle p\_4, skip\_3 \rangle$. Considering the dataset related to *p_3*, a new instance is added to the corresponding dataset, containing the most recent attributes values and transition *skip_2* as target.

Then, when reaching the *Check* activity in the variant, the dataset of decision point *p_3* is expanded with a new instance. This contains the newly updated attributes values and a target value of *Check*.

Tab. 3.1 shows how the dataset related to decision point *p_3* is filled according to the example variant. For the sake of this example, attributes values are selected randomly. Note that the *loan_accepted* attribute was not present in the Event Log when inserting the first instance in the dataset, and therefore that column was not present initially. Afterwards, when adding the second instance, which contains a valid value for that attribute, the *loan_accepted* column is inserted, filling previous entries with a missing value indicator, *?* in this case.

The pseudocode describing the decision points datasets construction procedure is reported in *Alg. 10*.

3.2.3 *Decision Trees training and rules extraction*

Each created dataset contains all the Event Log data related to a specific decision point inside the Petri Net model of the process. More precisely, every instance in the dataset refers to a distinct crossing of that decision point by some process execution; the instance is composed of the values of the process attributes as recorded in the Event Log at the time of the decision, and by a

Algorithm 10 Overall procedure to create decision points datasets.

```
Require: an Event Log and a Petri Net model of the same process
  procedure CREATEDPSDATASETS
      datasets ← initializeDatasets()
      variants ← getVariants(eventLog)
      for variant ∈ variants do
          variantMap ← ∅
          sequence ← ∅
          for activity ∈ variant do
              sequence ← addActivity(activity, sequence)
              if activity ≠ firstActivityInVariant then
                  transition ← getTransition(activity, net)
                  validDPs ← EXTRACTDPS(sequence, transition)
                  variantMap ← addDPs(activity, validDPs, variantMap)
              end if
          end for
          variantTraces ← getVariantTraces(variant, eventLog)
          for trace ∈ variantTraces do
              attr ← ∅
              for activity ∈ trace do
                  if activity ≠ firstActivityInTrace then
                      validDPs ← getDPs(activity, variantMap)
                      for (dp, trgt) ∈ validDPs do
                          datasets ← updtDataset(dp, trgt, attr, datasets)
                      end for
                  end if
                  attr ← updateAttrValues(attr, eventLog)
              end for
          end for
      end for
  end procedure
```

target attribute consisting of the name of the transition that has been followed after the decision point.

The structure of these datasets is ideal to solve the classification problem of Decision Points Analysis, since each is composed by a set of instances related to a single decision point, containing a set of attributes values and a discrete target value specifying the choice that has been made. Rules extraction is the goal, hence the Decision Tree approach proposed in the state-of-the-art is definitely fitting.

We rely on an implementation of the C4.5 algorithm [10] in order to fit a Decision Tree Classifier on each of the aforementioned datasets. Each fitted tree is then exploited to extract the guards of the output transitions of the related decision point, i.e., the rules which govern the routing of cases.

A common issue in fitting Decision Trees lies in the imbalance of the training set: a very skewed dataset (in terms of target values) may be problematic for the fitting procedure, such that the C4.5 algorithm may not be able to find any suitable split. A Decision Tree without any split cannot be used for rules extraction, since it does not represent any rule at all. This can happen frequently using the method proposed in this thesis, since a backward search only contains one visible activity and possibly many invisible ones. Hence, for a single pair of activities in a trace, only one of the valid decision points will contain the visible activity, but at the same time all the other valid decision points will contain invisible activities. Depending on the model, the probability of adding an invisible activity to the datasets is therefore much higher than adding a non-invisible one: this can lead to extremely unbalanaced datasets, up to two orders of magnitude.

A possible way to counter this issue resides in sampling techniques. Their goal is to rebalance the target class distribution in the dataset, and they achieve this by building a new dataset containing selected samples from the original one. When dealing with binary decision points, sampling can be performed in two opposite manners:

- *Oversampling*: new synthetic instances are generated starting from the examples belonging to the minority class, until there is a balance between the two classes. The simplest way to perform oversampling is by randomly duplicating instances of the minority class. More involved techniques, like *SMOTE* (*Synthetic Minority Oversampling TEchnique*), allow to create new instances that are close to the existing ones in the feature space. The advantage of oversampling methods is that there is no information loss, since all the existing instances in the dataset are still used. However, duplicating existing data can lead to overfitting, since a lot of examples would be in the exact same spot in the feature space; generating synthetic instances partially counters this issue, because the new samples are near existing ones.

- *Undersampling*: existing instances belonging to the majority class are selected until there is a balance between the two classes. As before, the straight way to perform undersampling is to randomly choose instances from the majority class. Alternative approaches has been developed, usually related to the *k-nearest-neighbors* algorithm, hence selecting instances from the majority class according to their distances to the ones of the minority class in the feature space. The advantage of undersampling methods is that the only exploited data is the one already present in the dataset, since there is no generation of new instances. The drawback is that some data is eliminated, since the goal is to reduce the number of examples related to the majority class.

Having evaluated advantages and disadvantages of the existing sampling methodologies, we opted for random undersampling. The goal of the algorithm proposed in Section 3.2.1 is to avoid information loss, hence multiple paths that could be followed during the process executions are selected, and the corresponding data is added to the datasets, which are often very large, as mentioned before. Given this abundance of information about possible decision points traversals, discarding some instances belonging to the majority class still results in rich datasets.

In conclusion, information is lost when performing undersampling in this context; however, the balanced datasets contain a lot more data with respect to previous methodologies, and this leads to finer results. Additionally, reducing the size of the dataset should help in avoiding excessive overfitting.

To support decision points which have more than two outgoing arcs (i.e., they are not binary), we decided to perform undersampling from the majority class for each of the minority classes. This way, the resulting dataset will be composed of the instances belonging to the minority classes, and a number of examples from the majority class that is equal to the sum of the instances of the minority classes. This way, the C4.5 algorithm can find a suitable split and the resulting Decision Tree can be used for rules extraction.

3.3 RULES PRUNING IMPLEMENTATION

As mentioned in Section 2.2.5, the rules extracted from a Decision Tree may be simplified in order to be more readable and understandable.

Among the two pruning approaches proposed earlier, the latter one (which directly simplifies guards) allows for possibly complete simplification of a production rule, meaning that an entire branch of the tree is removed. In particular circumstances, it may happen that all the production rules involving a specific `class c` are completely pruned from the tree. This results in a missing rule for that target class, therefore we would not be able to characterize that choice of the decision point with a logical expression.

To avoid this from happening, we can tune the only hyperparameter in this method, which is the threshold on the p-value. Indeed, increasing the threshold means that fewer production rules are pruned. However, finding a proper threshold value may be difficult, especially if we want this method to work on unseen processes. Hence, one could start from a low p-value threshold, for example 1%, and, if at the end of the process any rule is empty, the simplification procedure is repeated with an increased p-value threshold.

A 1% increase would result in finer pruning, at the expense of longer execution times. A much better idea would be to increase the p-value threshold by a quantity around 5% every time a target value is left without a rule.

3.4 ADAPTED IMPLEMENTATION OF THE DAIKON METHOD

As mentioned in Section 2.3.2, Daikon discovers several kinds of invariants, some of which are comparisons between:

- Continuous and categorical variables.
- Continuous and boolean variables.
- Categorical and boolean variables.
- Categorical variables.
- Boolean variables.

These kinds of comparisons are not often very interesting in the Decision Points Analysis field, especially since there is usually no point in comparing different types of variables. In [13], it is reported that *"Daikon supports three primitive data types (integer, float and string) and sequences these primitive data types"*. However, in our experience this support does not seem to be that straightforward. The inequalities do not appear to be based on lexicographical comparisons; instead, it looks like categorical and boolean variables are first converted to numerical ones through a mapping, and then the inequalities are computed and reported.

Some relations between non-continuous variables are apparently useful, but this does not hold in presence of missing values. As before, it looks like Daikon applies some mapping to non-continuous variables: a boolean variable taking values *True* and *False* will be mapped to *{0: False, 1: True}* but another boolean variable containing missing values *?* will be mapped to *{0: ?, 1: False, 2: True}* and therefore the resulting equality/inequality invariants would be misleading. It is worth mentioning that useful invariants related to categorical and boolean variables are reported by Daikon, e.g., *loan_accepted == "recheck"* or *is_present == True*, but they actually include only a single variable.

In general, the most interesting invariants reported by Daikon are the ones related to continuous variables. Indeed, it tests for unary, binary, and ternary invariants, and it can output invariants that are combination (linear and non-linear) of those variables. However, this does not always happen since it deeply depends on the initial dataset and on the attributes values. For example, given a dataset with only one continuous variable, Daikon may not be able to output a single invariant containing that variable.

Another aspect to discuss about the proposed approach is the final adjustment of conjunctive expressions. Although having two non-overlapping and mutually exclusive rules for the respective branches of the decision point makes sense in principle, it limits the results of the Decision Points Analysis process. Indeed, the conjunctive expressions obtained before the adjustment are based on process instances that followed the respective paths. With the final adjustment, we throw away one entire conjunctive expression between the two, putting it as the negation of the other one. This may lead to the loss of important information acquired during the rules discovery process. Additionally, process instances that follow the path of the discarded expression may not even have a value for the attributes specified by the negation of the other expression (because maybe those attributes were not useful in that process instance). In those cases, a check on the other condition is not even possible since those attributes are not even present.

Finally, avoiding the expressions adjustment allows to easily extend the approach also to non-binary decision points. Daikon discovers the invariants for each set of process instances related to each branch, and the final conjunctive expressions are built by greedily adding an invariant provided that the information gain increases, as explained before. Keeping the resulting expression as they are, may result in overlapping conditions, i.e., two (or more) branches are guarded by the same conjunctive expression. Daikon discovers invariants looking at the datasets independently and, even if the resulting expressions are built considering the information gain (hence also the datasets related to the other branches), the greedy approach may result in equal branching conditions. As presented in Section 2.3.1, this is something that can happen in the Decision Points Analysis field, and it usually reflects missing information in the Event Log since certain choices in business process may be the product of human decisions in presence of ambiguity, and not everything is recorded in the log.

One final remark about the approach presented in [13] is that the greedy selection of atoms to build the conjunctive expression may lead to non-deterministic results. Indeed, it may happen that two or more atoms have the same exact information gain. We can break this tie in different ways, for example choosing an alphabetical order of the atoms. Either way, this influences the outcome, i.e., the resulting conjunctive expression. The alternative guards are basically equivalent since the original atoms have the same information

gain. Nonetheless, if the attributes chosen to be in the final expressions end up being different depending on the tie-breaking rule, this could lead to potentially different outcomes while testing on new instances, since we test on completely different variables. An alternative approach could be to put in disjunction the atoms with the same information gain; however, this would most likely result in very long resulting expressions.

Among the approaches presented in [13], we implemented the *CD+IG+LV* variant, which is the discovery of conjunctive conditions using Daikon and the greedy approach based on information gain, all of which using the dataset enriched with latent variables. As shown in *Alg. 11*, the major differences with respect to the proposed method are linked to the elimination of some of the invariants discovered by Daikon (the ones which do not really make sense, as reported before) and the final conditions adjustment. Indeed, this is done only if the analyzed decision point is binary, rewriting the rule as *not (rule)* if it is a conjunction of multiple invariants, or simply negating the operand if the rule is composed by only one invariant. Additionally, the input dataset is preprocessed as follows: instances containing missing values for continuous attributes are removed, while the other missing values are replaced with *?*. Finally, when picking the invariant with the maximum information gain, the tie-breaking rule selects the atom which comes last alphabetically.

Algorithm 11 Our implementation of the CD+IG+LV variant using Daikon.

```
procedure discoverRulesWithDaikon(trainingSet)
    expressions ← ∅
    for target ∈ trainingSet do
        setTarget ← getSetTarget(trainingSet, target)
        setTarget ← addLatentVariables(setTarget)
        invariants ← getDaikonInvariants(setTarget)
        invariants ← removeMeaninglessInvariants(invariants)
        conjExpr ← buildConjExpr(invariants, trainingSet)
        expressions ← addExpr(expressions, conjExpr)
    end for
    if isBinaryDecisionPoint(trainingSet) then
        expressions ← adjustConditions(expressions)
    end if
end procedure

function removeMeaninglessInvariants(invariants)
    for invariant ∈ invariants do
        contNonCont ← isCompContAndNonCont(invariant)
        nonConts ← isCompBetweenNonConts(invariant)
        if contNonCont ∨ nonConts then
            invariants ← removeInvariant(invariant)
        end if
    end for
    return invariants
end function

function buildConjExpr(invariants, trainingSet)
    resultingExpr ← ∅
    remainingInvariants = invariants
    while remainingInvariants ≠ ∅ do
        maxGainInv ← getInvWithMaxIG(invariants, trainingSet)
        newExpr ← combineExprs(resultingExpr, maxGainInv)
        newIG ← computeIG(newExpr, trainingSet)
        if newIG > computeIG(resultingExpr, trainingSet) then
            resultingExpr = newExpr
        end if
        remainingInvariants ← removeInv(maxGainInv)
    end while
    return resultingExpr
end function
```

4

EVALUATION

4.1 PROCESSES

In order to test the novel method proposed in this thesis, we relied on two distinct processes: a synthetic one and a real one. The former is the running example presented in Section 3.1, whose Event Log is generated by arbitrarily selecting attributes values (respecting the guards) and by firing transitions at random. We generated a total number of 100 traces, which can be grouped in 31 different variants. The activities belong to the set $\langle$*Check, FinalCheck, NOK, OK, PrepDoc, Register, Request*$\rangle$ as previously shown. There are five additional different attributes: *amount* is a continuous one, *loan_is_accepted* is a categorical one having *yes, no* and *recheck* as possible values, while *doc_is_updated, is_present, loan_is_accepted* and *skip_everything* are boolean ones.

The latter is instead an Event Log stored by an information system of an Italian municipal police force, which was introduced in [14]. This log is composed of 150370 traces, divided in 231 variants, ranging from a length of 2 activities up to 20 activities. There is a total of 11 different activities, belonging to the set $\langle$*Add Penalty*, *Appeal to Judge*, *Create Fine*, *Insert Date Appeal to Prefecture*, *Insert Fine Notification*, *Notify Result Appeal to Offender*, *Payment*, *Receive Result Appeal from Prefecture*, *Send Appeal to Prefecture*, *Send Fine*, *Send for Credit Collection*$\rangle$.

This real-life process contains 13 additional distinct attributes: *amount*, *expense*, *paymentAmount*, *points* and *totalPaymentAmount* are continuous ones, *article*, *dismissal*, *lastSent*, *matricola*, *notificationType*, *org:resource* and *vehicleClass* are categorical ones, and there is an extra variable *time:timestamp* containing the time at which the activity has been executed.

Given these two Event Logs, we extracted the related Petri Nets through the Inductive algorithm, in order to also discover invisible transitions. More precisely, to obtain the running example net we relied on an implementation of the Inductive Miner inside the PM4PY library [15] for the *Python* programming language. Instead, the net related to the real-life Event Log has been discovered using an implementation of the Inductive Miner in the ProM framework, so that the model is identical to the one analyzed in [8].

The different implementations of the Inductive Miner allow to specify a noise threshold, ranging from 0 to 1. In particular, a noise threshold of 0 means that the discovered net perfectly represents the information contained in the Event Log. Increasing the noise threshold value, the extracted net will

ignore some behaviors recorded in the log: therefore, the obtained model will not be completely sound with respect to the Event Log, and certain variants may not be perfectly replicated. We extracted both the Petri Nets using a noise threshold of 0, in order to have perfectly sound models.

The mined real-life process Petri Net contains different parallel branches, starting from the ones splitting from transition *Create Fine* at the very beginning. Moreover, the bottom branch also contains a loop on activity *Payment*, which can be possibly fired many times consecutively, choosing the invisible transition *skip_7* after decision point *p_12*. The loop is terminated whenever the decision goes towards transition *skip_8*. Notice that the loop, and therefore transition *Payment*, could also be entirely skipped passing through the invisible transition *skip_4*.

It is interesting to observe that, increasing the noise threshold of the Inductive Miner to a value of *0.2*, the loop is not detected anymore. Transition *Payment* can be executed once, or it can be skipped as before.

Fig. 4.1 shows the Petri Net of the real-life process, extracted using the Inductive Miner implemented in ProM with a noise threshold of 0.

4.2 RESULTS

In this section we present and discuss the results produced by the method proposed in this thesis on the two datasets introduced before. Concerning the real-life process, we also compare the results with the ones presented in [8], since both the Event Log and the Petri Net model are identical. For Decision Trees construction, our method relies on an implementation of the C4.5 algorithm which supports all types of attributes; an additional parameter imposes a maximum depth of the trees (i.e., the maximum number of levels in the tree) of 7, in order to avoid gigantic structures.

4.2.1 *F1-score measure*

In order to quantitatively measure the quality of the discovered guards, for each decision point we tested the related Decision Tree with the corresponding dataset extracted using the novel method. We then measured the accuracy of the predictions through the *F1-score*, which is related to both the *precision* and the *recall* of the test.

Considering a binary dataset, that is a dataset with only two possible target values, we define one of them as the positive class and the other as the negative one. Precision is defined as the ratio between the number of true positives (*tp*), i.e., members of the positive class that are correctly predicted as positive, and the sum of true positives and false positives (*fp*), the latter ones being the instances of the negative class that are mistakenly predicted as

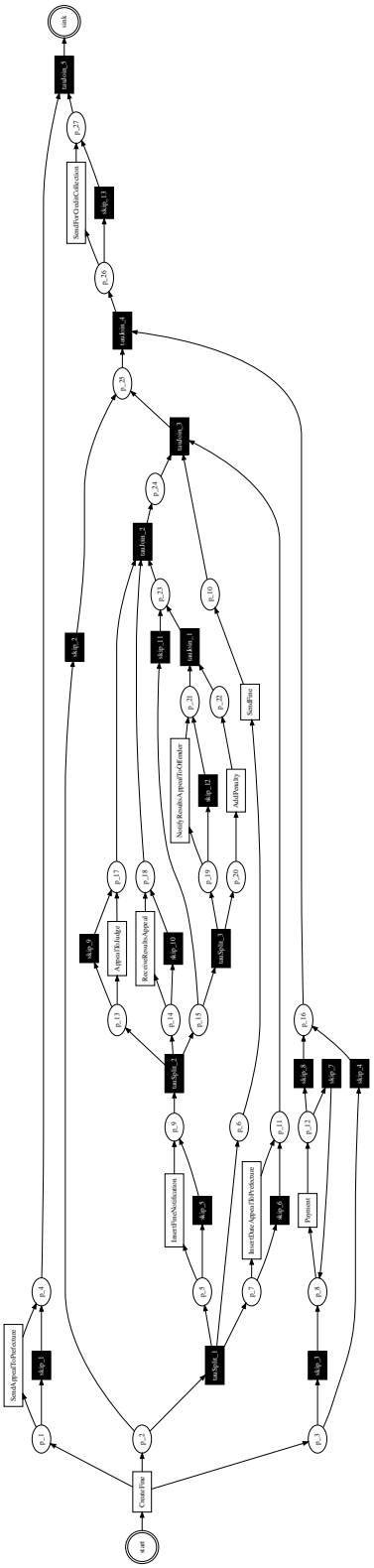


Figure 4.1: Petri Net of the real-life process

Decision Point	Found	F1-score	Rule Discovered
p_1	yes	1	exact
p_3	yes	0.7801	partially correct
p_4	yes	0.9719	exact
p_8	yes	1	exact

Table 4.1: Results applying the proposed method to the running example process

positive. Hence, precision shows how many instances are correctly classified as positive among the ones that are predicted as positive.

The recall of a prediction is instead defined as the ratio between the true positives and the sum of true positives and false negatives (*fn*), the latter ones being the members of the positive class that are incorrectly predicted as negative. In other words, recall represents how many instances are classified as positive among the ones that are actually positive in the dataset.

Finally, the F1-score is defined as two times the product between precision and recall, divided by the sum of these two quantities, as shown below in 4.1.

$$\text{Precision} = \frac{tp}{tp + fp} \quad \text{Recall} = \frac{tp}{tp + fn} \quad \text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.1}$$

When dealing with non-binary datasets and therefore more than two possible target values, the F1-score is computed for each class separately. The values can then be combined in different ways: we opted for a weighted average between them, considering the number of instances belonging to each class. In the two test processes evaluated in this thesis, the only non-binary decision point is *p_8* in the running example Petri Net, which has three outgoing arcs.

4.2.2 *Running example process*

Tab. 4.1 shows the results concerning the running example process. The proposed method is able to discover the guards for all the output transitions of all the decision points inside the Petri Net model. Each decision point dataset is balanced in terms of target values, therefore there is no need to apply an undersampling technique since the C4.5 algorithm is able to find suitable splits. The extracted rules are almost always identical to the actual ones, and they are reported in *Tab. 4.2*.

The only slight difference resides in the output guards for decision point *p_3*. Indeed, the attribute *loan_is_accepted* is not part of the discovered rules,

Decision Point	Target	Discovered Rule
p_1	Register	$is\_present = False$
	skip_1	$is\_present = True$
p_3	Check	$(skip\_everything = False \wedge amount > 398) \vee$
		$(skip\_everything = True \wedge amount > 987)$
	skip_2	$(skip\_everything = False \wedge amount \leqslant 398) \vee$
		$(skip\_everything = True \wedge amount \leqslant 987)$
p_4	PrepDoc	$doc\_is\_updated = False \wedge$
		$skip\_everything = False$
	skip_3	$skip\_everything = True \vee$
		$(doc\_is\_updated = True \wedge$
		$skip\_everything = False)$
p_8	OK	$loan\_is\_accepted = yes$
	NOK	$loan\_is\_accepted = no$
	skip_4	$loan\_is\_accepted = recheck$

Table 4.2: Discovered rules applying the proposed method to the running example process

while it was there in the original Data Petri Net. To understand the reason behind this, we have to analyze the fitted Decision Tree: in fact, there is no split on attribute *loan_is_accepted*, because it only takes the value *recheck* or it is missing.

Indeed, a process execution traverses *p_3* for sure once, either to execute transition *Check* or to skip that activity passing through the invisible transition *skip_2*. On this first pass, attribute *loan_is_accepted* is not defined, since it takes a value only after the activity *FinalCheck*. Therefore, the decision point dataset would contain a missing value indicator for this crossing.

However, if the same process execution performs a loop, then it traverses again decision point *p_3*, but this time the attribute *loan_is_accepted* is perfectly defined, having value *recheck*. As a matter of fact, a process instance can loop only if activity *FinalCheck* signals that there is the need to recheck the loan request.

In conclusion, when considering *p_3*, the attribute *loan_is_accepted* can only be a missing value or equal to *recheck*. Since the C4.5 splitting algorithm of the Decision Tree only considers non-missing values when computing the information gain, it only observes value *recheck*. This results in an information gain of zero, hence no splitting and, therefore, no condition on attribute *loan_is_accepted* along the branches of the tree.

Decision Point	Target	Discovered Rule
p_1	Register	is_present = False
	skip_1	is_present = True
p_3	Check	(skip_everything = False ∧ amount > 398)
	skip_2	(skip_everything = True ∨ amount ⩽ 398)
p_4	PrepDoc	doc_is_updated = False ∧ skip_everything = False
	skip_3	doc_is_updated = True ∨ skip_everything = True
p_8	OK	loan_is_accepted = yes
	NOK	loan_is_accepted = no
	skip_4	loan_is_accepted = recheck

Table 4.3: Simplified guards applying the production rules pruning method to the running example process

Considering decision point *p_4*, we observe that the discovered rules are perfectly in line with the ones in the original Data Petri Net; however, the F1-score is slightly less than 1. The reason behind this result is to be traced back to what has been described in Section 3.2.1: the proposed depth-first search algorithm is not able to detect decision points on branches that are parallel to the transition it has to reach.

For this reason, in such particular conditions the method does not add the related instance in the dataset of *p_4*, therefore losing some data. When performing a prediction, the resulting accuracy cannot be exactly 1, since those cases have not been covered. An analogous behavior affects decision point *p_3*, whose F1-score is already lower than 1 given the rationale explained before.

Applying the rule production pruning method introduced in Section 2.2.5, allows to obtain even more compact rules. Interestingly, the guards for the output transitions of *p_4* become the exact copy of the ones in the original Data Petri Net, and rules related to decision point *p_3* now are also more similar to the original ones. The simplified rules are shown in *Tab. 4.3*.

On the other hand, the pessimistic pruning approach presented in Section 2.2.5 does not seem to have any effect on the extracted rules. This is reasonable, since guards related to the running example process are already short and straightforward.

The computation of overlapping rules as explained in Section 2.3.1 allows to add further conditions in disjunction with the ones already found. More

precisely, the guard for transition *Check* is expanded with the following conditions:

$$\texttt{skip\_everything} = \texttt{True} \wedge \texttt{amount} \leqslant 987$$

$$\texttt{skip\_everything} = \texttt{False} \wedge \texttt{amount} \leqslant 398$$

Which correspond exactly to the guard for transition *skip_2* of the same decision point *p_3*. This indicates that many instances having transition *Check* as target are misclassified by the Decision Tree, since their actual target is *skip_2*. This is in line with the recorded F1-score for this decision point of 0.7801.

Additionally, for the same reason, the guard for transition *skip_3* is expanded with the corresponding condition obtained for transition *PrepDoc*, that is:

$$\texttt{doc\_is\_updated} = \texttt{False} \wedge \texttt{skip\_everything} = \texttt{False}$$

For the sake of completeness, we also comment the guards discovered using the Daikon invariant detector method presented in Section 2.3.2, discarding continuous attributes with missing values and meaningless invariants. Daikon is not able to find guards for outgoing transitions of decision points *p_1*, *p_3* and *p_4*. Instead, it discovers the correct rules related to *p_8*.

In conclusion, at least concerning the running example process, applying the rule production simplification method to the guards extracted from the Decision Trees seems to provide the best results in term of readability and faithfulness to the logical conditions in the original Data Petri Net.

4.2.3 *Real-life process*

We found the most interesting results applying the proposed method to the Event Log related to the real-life process, mostly because of its larger size and variety of cases. Besides, being able to compare the results with the ones presented in [8] allows to show the improvements introduced by the method itself.

Tab. 4.4 shows a comparison between our method and the state-of-the-art one, which makes use of optimal alignments. For every decision point in the Petri Net model, it is reported whether or not each method was able to find the guards for their outgoing transitions, along with the corresponding F1-score.

It is immediately visible that our method is able to discover guards for all the decision points in the given model, while the optimal alignments approach only managed to get rules for 5 out of the 11 decision points. On top of that, the F1-scores are always higher, indicating a better prediction accuracy of the learned Decision Tree Classifiers.

The extracted rules are therefore more precise, with the downside that they are lengthy for certain output transitions. This is not an issue if a machine has

	Proposed Method		Optimal Alignments	
Decision Point	Found	F1-score	Found	F1-score
p_1	yes	0.791	yes	0.755
p_2	yes	0.908	yes	0.458
p_3	yes	0.766	yes	0.567
p_5	yes	0.833	no	0.434
p_7	yes	0.892	no	0.49
p_12	yes	0.900	yes	0.808
p_13	yes	0.892	no	0.499
p_14	yes	0.954	no	0.498
p_15	yes	0.692	no	0.434
p_19	yes	0.971	no	0.497
p_26	yes	0.968	yes	0.933

Table 4.4: Comparison between the results of the proposed method and the optimal alignments one on the real-life process

to interpret them, but it may become one if we consider human-readability. Indeed, some of the guards contain tens of conditions in disjunction, resulting in extended rules which are difficult to understand.

This problem is partially mitigated by rule pruning methodologies, that are able to eliminate a few conditions. However, certain guards still remain far too long to be easily read and understood. A possible further help may come from modifying the algorithm for Decision Tree learning, for example reducing the maximum depth of the tree, or imposing a minimum number of instances for each leaf, in order to obtain smaller trees and, consequently, shorter rules.

In order to test our method on incomplete models, i.e., Petri Nets that do not faithfully represent all the possible variants contained in the Event Log, we also extracted the model for this real-life process using the Inductive Miner and specifying a noise threshold of 0.2. The resulting Petri Net is identical to the one studied so far, with the exclusion of the loop: place *p_3* has two outgoing arcs, one toward *Payment* and one toward *skip_4*; then, transition *Payment* goes directly into place *p_16*, since there is no loop.

The final version of the proposed algorithm is able to manage such situations. Consider the sequence ⟨*Create Fine, Payment, Payment*⟩: the depth-first search would start from the last *Payment* activity, following invisible transitions backward until it reaches the first *Payment* activity. However, since the

loop is not present anymore in the model, this is not possible. Therefore, the algorithm selects the previous activity *Create Fine* as the one to be reached.

This means that place *p_3* has been chosen twice in the sequence, even if it is not part of a loop. Indeed, it is selected on the backward path between the first *Payment* and *Create Fine*, but also on the one between the second *Payment* and the first *Payment*. Hence, the dataset of *p_3* would contain two different instances related to these two traversals.

As mentioned in the previous chapter, this is not an issue, since the same process execution passed through that decision point in order to reach *Payment* the first time and, according to the Petri Net model, reaching *Payment* a second time would mean crossing *p_3* again. This is also supported by the results, since the F1-scores for this incomplete net are identical to the ones obtained before.

In general, the datasets related to many of the decision points are very unbalanced in terms of target values, such that the C4.5 algorithm is not able to find any suitable split in order to build a Decision Tree. As mentioned in the previous chapter, this issue is amplified by the proposed method, since for each visible transition on a path, many invisible ones are taken into consideration, resulting in unbalanced datasets.

Therefore, for this real-life process a rebalancing technique is needed in order to extract rules for more than three decision points (*p_3*, *p_5*, *p_26*) output transitions. As proposed earlier, we relied on an undersampling technique in order to obtain perfectly balanced datasets. This allows to build a proper Decision Tree Classifier for each decision point dataset, hence extracting related guards.

Given that random undersampling may lead to slightly different results every time the sampling is executed, the F1-score values reported in *Tab. 4.4* are average values. More precisely, we decided to perform undersampling, Decision Tree fitting and F1-score computation 10 times, averaging the resulting quantities in order to obtain a more realistic value.

On the other hand, the Daikon invariant detector approach does not suffer from unbalanced datasets, since it considers each subset of instances with the same target value and discovers invariants that are valid in that set. Concerning this real-life process, our adapted implementation of Daikon presented in Section 2.3.2 is able to find guards for almost all decision points. The extracted conjunctive conditions are more compact with respect to the ones discovered with Decision Trees, and they often contain linear combination of continuous attributes.

For instance, considering decision point *p_1* and target transition *skip_1*, Daikon discovers the following guard:

$$\mathtt{totalpaymentAmount} \cdot \mathtt{points} \geqslant \mathtt{points} \cdot \mathtt{paymentAmount}$$

A more complex rule is instead found when considering decision point p_5 and its target transition *Insert Fine Notification*. Note that it has been slightly rewritten to manually factor out some terms:

$$2 \cdot (expense - totalPaymentAmount) = 0 \wedge -2 \cdot (amount + points) = 0$$

In conclusion, concerning the real-life process, the novel method proposed in this thesis is able to discover the guards for the output transition of all the decision points in the Petri Net model, something that the optimal alignment approach was not able to do. Moreover, the recorded F1-scores are higher, which translates in better Decision Tree Classifiers for prediction and finer rules.

However, not all the guards are easily readable by humans because of their excessive length. The two rule pruning methods that have been tested are not always able to simplify the guards enough to be comfortably intelligible. The approach relying on the Daikon invariant detector is instead able to extract more compact rules, although it cannot discover guards for all the decision points.

The results presented in this section do not aim to represent an extensive proof of the general validity of the proposed method. Nonetheless, they are encouraging, as the comparison with the state-of-the-art technique using optimal alignments shows several advantages of this novel approach.

5

CONCLUSIONS AND FUTURE WORKS

In this thesis we proposed a novel method to perform Decision Points Analysis on a Petri Net model of a business process discovered starting from an Event Log. The approach differs from the state-of-the-art ones because it exploits more information related to the control-flow structure of the model. More precisely, current methodologies either follow the path of the process instance until an ambiguous place is found, or they choose a single optimal path in the entire net.

Our aim is to exploit all the possible paths that a process instance may follow according to the corresponding Petri Net model, traversing invisible transitions. Unfortunately, we did not manage to produce an algorithm able to correctly generalize on all the possible control-flow structures, since in some cases it would also consider incorrect paths. Therefore, we developed a technique which takes into consideration all the possible paths in most of the cases, in order to avoid considering invalid ones in particular conditions.

Nonetheless, this method is able to keep track of more paths with respect to current techniques. This knowledge is then exploited in order to perform Decision Points Analysis, considering each decision point in the Petri Net as a classification problem, as already proposed in the state-of-the-art methods. These classification problems are solved using Decision Tree learners, which are then employed to extract the guards of the decision points output transitions.

The results on both a synthetic and a real process show that the proposed method is indeed exploiting more information, since it discovers guards for transitions where previous approaches failed. Additionally, the quality of the extracted guards is higher, as proved by the F1-scores. This allows for better process analysis and prediction capabilities, two key aspects when dealing with business processes.

Reducing information loss is one of the main challenges in Decision Points Analysis and future works should follow that direction. The proposed method could be extended in order to consider all possible paths in every situation, especially when dealing with unexplored parallel branches. This would most likely result in even higher F1-scores, hence accurate models for prediction, since less knowledge is left out.

Future research should also delve into the two main issues mentioned in this thesis: imbalanced datasets and lengthy rules. The former is an intrinsic consequence of considering all possible paths, and alternative rebalancing techniques should be studied with the aim of avoiding losing too much

information but at the same time averting overfitting. The latter could be mitigated either stopping the Decision Tree fitting procedure earlier, obtaining smaller trees, or exploiting more advanced rule pruning techniques. This is a crucial aspect, since shorter guards are more readable by humans, which is necessary for easier process diagnostic and enhancement.

BIBLIOGRAPHY

[1] Tom M. Mitchell. "Machine Learning and Data Mining." In: *Communications of the ACM* 42 (1999), pp. 30–36 (cit. on p. 1).

[2] Wil Aalst, Arya Adriansyah, Ana Medeiros, Franco Arcieri, Thomas Baier, Tobias Blickle, Jagadeesh Chandra Bose R.P., Peter Brand, Ronald Brandtjen, Joos Buijs, Andrea Burattin, Josep Carmona, Malú Castellanos, Jan Claes, Jonathan Cook, Nicola Costantini, Francisco Curbera, Ernesto Damiani, Massimiliano de Leoni, and Moe Wynn. "Process Mining Manifesto." In: vol. 99. Aug. 2011, pp. 169–194. ISBN: 978-3-642-28107-5 (cit. on pp. 1, 9).

[3] Peter Wong. "Compositional Development of BPMN." In: vol. 8088. June 2013, pp. 97–112 (cit. on p. 2).

[4] A. Rozinat and W. M. P. van der Aalst. "Decision Mining in Prom." In: *Proceedings of the 4th International Conference on Business Process Management*. BPM'06. Vienna, Austria: Springer-Verlag, 2006, 420–425. ISBN: 3540389016 (cit. on pp. 3, 4, 20, 27).

[5] A. Rozinat and W.M.P. Aalst, van der. *Decision mining in business processes*. BETA publicatie : working papers. Technische Universiteit Eindhoven, 2006. ISBN: 90-386-0685-0 (cit. on pp. 3–5).

[6] Wil van der Aalst, Ton Weijters, and Laura Maruster. "Workflow Mining: Discovering Process Models from Event Logs." In: *IEEE Trans. on Knowl. and Data Eng.* 16.9 (2004), 1128–1142. ISSN: 1041-4347 (cit. on p. 3).

[7] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. "Discovering Block-Structured Process Models from Event Logs - A Constructive Approach." In: *Application and Theory of Petri Nets and Concurrency*. Ed. by José-Manuel Colom and Jörg Desel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 311–329. ISBN: 978-3-642-38697-8 (cit. on p. 3).

[8] Massimiliano de Leoni and Wil M. P. van der Aalst. "Data-Aware Process Mining: Discovering Decisions in Processes Using Alignments." In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. SAC '13. Coimbra, Portugal: Association for Computing Machinery, 2013, 1454–1461. ISBN: 9781450316569 (cit. on pp. 6, 20, 27, 47, 48, 53).

[9] Gustavo Callou, Paulo Maciel, Dietmar Tutsch, Julian Araújo, João Ferreira, and Rafael Souza. "A Petri Net-Based Approach to the Quantification of Data Center Dependability." In: *Petri Nets*. Ed. by Pawel Pawlewski. Rijeka: IntechOpen, 2012. Chap. 14 (cit. on p. 10).

[10] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 1558602380 (cit. on pp. 12, 15, 40).

[11] J. R. Quinlan. "Simplifying Decision Trees." In: *Int. J. Hum.-Comput. Stud.* 51.2 (1999), 497–510. ISSN: 1071-5819 (cit. on pp. 15, 17).

[12] Felix Mannhardt, Massimiliano de Leoni, Hajo A. Reijers, and Wil M. P. van der Aalst. "Decision Mining Revisited - Discovering Overlapping Rules." In: *Advanced Information Systems Engineering*. BPM Center Report. Springer International Publishing, 2016, pp. 377–392. ISBN: 9783319396965 (cit. on p. 19).

[13] Massimiliano de Leoni, Marlon Dumas, and Luciano García-Bañuelos. "Discovering Branching Conditions from Business Process Execution Logs." In: *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*. FASE'13. Rome, Italy: Springer-Verlag, 2013, 114–129. ISBN: 9783642370564 (cit. on pp. 20, 22, 23, 42–44).

[14] F. Mannhardt, M. Leoni, de, H.A. Reijers, and W.M.P. Aalst, van der. *Balanced multi-perspective checking of process conformance*. BPM Center report. BPMcenter. org, 2014 (cit. on p. 47).

[15] Alessandro Berti, Sebastiaan J. van Zelst, and Wil van der Aalst. *Process Mining for Python (PM4Py): Bridging the Gap Between Process- and Data Science*. 2019 (cit. on p. 47).