A Comparative Study of Process Mining Tools

FlexFringe, ProM, MINT and PRINS

M.L. Kloppenburg

Master's Thesis
Delft, University of Technology
Faculty of Electrical Engineering, Mathematics & Computer Science

A Comparative Study of Process Mining Tools

FlexFringe, ProM, MINT and PRINS

by

M.L. Kloppenburg

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on October 26th, 2022 at 1 PM.

Student number:	4383265	
Project duration:	January 1, 2022 – October 22, 2022	
Thesis committee:	Prof. dr. ir. S. E. Verwer,	TU Delft, supervisor
	Dr. J. H. Krijthe	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft Delft University of Technology

Acknowledgements

During this thesis, I learned a lot about both the subject matter and my personal way of working. My supervisor, Sicco Verwer, was a great help during my thesis. Especially, during the exploratory phase, which I struggled with. Furthermore, I'd like to thank Chris Hammerschmidt for helping me to get FlexFringe running on Windows. I also appreciate the other students that were there during the group meetings and were available to talk to every once in awhile.

Most of all, I would like to thank my parents, who have always supported me. They helped me through hard times and always kept helping me even when study progress was very slow. Lastly, I want to thank my boyfriend, who provides a relaxing and calm influence when I am stressed.

M.L. Kloppenburg
Delft, October 2022

Abstract

Nowadays, software is an integral part of many companies. However, the codebase can grow large and complicated and is often insufficiently documented. To gain insight, tools have been made to infer state machines and process models from software logs. These tools produce different types of models such as automata and Petri nets. The main objective of this research is to determine which tool is the optimal choice for inferring a readable and correct model within reasonable time. Currently, Petri nets and automata are not compared to each other and not all key performance indicators are applicable to both model types. To compare these different concepts, suitable metrics must be identified.

For this work, 8 configurations of 4 programs will be compared. Finite State Machines (FSMs) will be inferred with FlexFringe (AIC), MINT and PRINS (using MINT internally). Petri nets will be mined with ProM using the Inductive Miner, Inductive Miner Infrequent - All Operators, Hybrid-ILP and the Directly Follows miner. The configurations will use 5-folds cross validation to infer models on 9 software logs. Negative traces were synthesised as they were not available. The quality of the models will be measured though inference time, complexity, F_2-score, balanced accuracy, fitness and perplexity.

Some of the used metrics were adequate, but others were not suitable. Inference time, F_2-score, balanced accuracy could be measured for both FSMs and Petri nets. The complexity was measured with the Petri net $eCFC$ metric and the Cyclomatic complexity CC. The $eCFC$ does not properly express complexity on FSMs. Furthermore, Petri nets can model parallelism, which introduces extra complexity compared to an FSM. This was not adequately expressed by either of these metrics. To measure fitness, both token-based replay and alignment fitness were used. FSMs were converted to Petri nets. Token-based replay (TBR) fitness was not an expressive metric for the FSMs, as the concept of tokens did not carry over well. In addition to this, the external implementation of TBR fitness was flawed for the specific structure of the converted FSMs. Alignment-based fitness is the superior fitness metric as it does not rely on the notion of tokens, which the FSMs do not have. Unfortunately, the time and memory needed for alignment computations was too large for some models. Lastly, the perplexity FSM metric was successfully adapted for Petri nets. It expresses the difference in structure and could be tailored even further for the purposes of comparison by adjusting its parameters.

The results of the comparison showed that almost all configurations could complete inference in feasible time and memory. The time out was set at 4 hours and the available memory was `16GB`. The MINT and PRINS ran out of memory on one of the larger logs and timed out for one other set. Hybrid-ILP timed out for 2 sets. All other configurations completed inference for all data sets in under 40 seconds. PRINS and MINT boasted excellent performance across all correctness metrics, and were only outperformed by FlexFringe and the Directly Follows miner on perplexity. PRINS and MINT were most suitable for modelling traces of data sets with a low trace similarity and generalising to identify new traces. However, MINT models were a lot larger than those of any other configurations and PRINS models are generally many times larger than MINT models. So, if complexity of models is a big concern, FlexFringe and the Directly Follows miner offer the smaller models, at the cost of a small amount of performance for most sets. However, these two tools perform poorly on sets that are both incomplete and contain dissimilar traces. If time is of the essence, one should use FlexFringe, the Directly Follows miner of one of the Inductive Miners. The Petri net miners used, were not designed to introduce new behaviour in a model. Therefore, FlexFringe is preferable for modelling an incomplete log.

Contents

1 Introduction **1**
1.1 Research Objectives . . . 2
1.2 Thesis Structure . . . 2

2 Background **3**
2.1 Software Execution Traces. . . . 3
2.2 Finite State Machines . . . 3
2.3 Petri Nets . . . 5
2.3.1 Conversion to Petri Net from FSM. . . . 6
2.4 Model Inference Techniques . . . 6
2.4.1 Directly Follows Miner . . . 6
2.4.2 Hybrid ILP Miner . . . 7
2.4.3 Inductive Miner . . . 8
2.4.4 Evidence-Driven Blue-Fringe State-Merging . . . 10
2.4.5 MINT . . . 11
2.4.6 PRINS. . . . 13
2.5 Evaluation Techniques . . . 13
2.5.1 F-score . . . 14
2.5.2 Balanced Accuracy. . . . 14
2.5.3 Fitness . . . 14
2.5.4 Perplexity . . . 15
2.5.5 Complexity . . . 16

3 Related Work **19**
3.1 ProM miners . . . 19
3.1.1 Process Discovery Contest 2021 . . . 19
3.1.2 Hybrid ILP miner . . . 19
3.1.3 Inductive Miners . . . 20
3.2 FlexFringe . . . 20
3.3 MINT . . . 21
3.4 PRINS. . . . 21

4 Methodology **23**
4.1 Tools and Settings . . . 23
4.1.1 Model Inference . . . 23
4.1.2 Evaluation Tools . . . 24
4.2 Data Sets . . . 24
4.2.1 Data Preparation . . . 25
4.2.2 Data Analysis . . . 25
4.3 Model Analysis . . . 26
4.3.1 k-Folds Cross-Validation . . . 26
4.3.2 Parsing and Conversion . . . 26
4.3.3 Synthesising Negative Logs . . . 26
4.3.4 General Metrics. . . . 26
4.3.5 Token-Based Fitness . . . 27
4.3.6 Alignment-Based Fitness and Cost . . . 27
4.3.7 Perplexity . . . 27
4.3.8 Cyclomatic and Control Flow Complexity . . . 29
4.3.9 Run Time . . . 29

4.4 Evaluation . . . 30
4.4.1 Data Sets . . . 30
4.4.2 Data Analysis . . . 30
4.4.3 Inference Method and Parameters . . . 30
4.4.4 Negative Log Synthesis . . . 31
4.4.5 Run Time . . . 31
4.4.6 Comparing Petri Nets to FSMs . . . 32

5 Results **35**
5.1 Run Time . . . 35
5.2 Complexity of Models . . . 36
5.3 Correctness of Models . . . 38
5.3.1 Soundness . . . 38
5.3.2 F_2-score and Balanced Accuracy . . . 38
5.3.3 Fitness . . . 41
5.3.4 Perplexity . . . 43
5.4 Data Composition and Performance . . . 44

6 Discussion **47**
6.1 Comparing Apples to Oranges . . . 47
6.1.1 Complexity . . . 47
6.1.2 Trace Fitness . . . 49
6.1.3 Perplexity . . . 50
6.2 Correctness of Models . . . 51
6.2.1 Soundness . . . 51
6.2.2 Overfitting and Generalisation in FSMs . . . 52
6.2.3 Recall and Specificity for ProM Models . . . 53
6.2.4 Precision . . . 54
6.2.5 Alignment Fitness . . . 54
6.2.6 Perplexity . . . 55
6.2.7 Conclusion . . . 55
6.3 Model Complexity and Performance . . . 56
6.4 Inference Time and Performance . . . 56
6.5 Data Set Characteristics Influence . . . 56
6.6 Limitations . . . 57
6.6.1 Parameter Tweaking . . . 57
6.6.2 Time and Memory . . . 57
6.6.3 ProM CLI . . . 57
6.6.4 Data Sets . . . 57

7 Conclusion **59**

A Model inference tool settings **67**
A.1 FlexFrings run settings . . . 67
A.2 ProM run settings . . . 67

B Pseudo code **69**
B.1 Perplexity for Petri nets . . . 69
B.2 Token-based replay for graphs . . . 70

C Full results **71**

D Data Composition and Performance **75**

E Generated Models **83**

1
Introduction

Nowadays, computers with their software are an integral part of many companies. Software is often used to perform many different types of complicated tasks. Thus, a piece of software can get large and difficult to understand. As needs and requirements for the software change over time, the company may start adjusting the software and adding new features to it. This may prove challenging.

First of all, the developers that originally created the software may no longer be at the company. The new developers will need correct, up-to-date and complete documentation if they wish to add components to the software, without causing bugs, in a timely manner. In reality, documentation is often not up-to-date, badly written or unclear, incorrect or just difficult to find [30].

Secondly, software will deteriorate over time [37], further increasing the need for documentation. One factor to deterioration occurs when adding components. A new developer may lack of understanding of the software's intended structure and make a change incompatible with it. If this happens many times over a span of years, the software structure will degrade and ultimately, there will be no one left that understands its true intent. When software deteriorates, its performance and reliability decreases [37], costing the company time and money.

To solve the problem of missing documentation and prevent further software deterioration, the new developers may wish to identify and analyse the structure and behaviour of the software before starting work on it. This can be done through *process mining* and *inferring state machines*. This produces a visual model that represents observed software behaviour. Process mining and state machine inference are two separate fields of research that are able to produce models for software traces. Research into inferring state machines has been done for decades and has produced many different methods[4][10][51][14][12]. Process mining has been around since the 90s and has also spawned many different approaches to process discovery [47]. Many of these inference and process discovery methods have been implemented in tools that can be used by anyone.

The question that arises now is, which tool and which method should a developer choose? This question unfortunately does not have straightforward answer. The creators of the aforementioned methods analyse and compare the performance of the models. But the models created by process mining are not compared to those produced by state machine inference methods. Process models and state machines differ in concepts, thus invalidating certain metrics commonly used on either process models or state machines. To bridge this gap, it is important to determine *how* a comparison be made between these different concepts.

Let us demonstrate this problem with a simple example. A developer has logged the behaviour of its software which shows that the software can call 5 difference methods a, b, c, d, e in 3 different orders: $\{a, b, c, d\}$, $\{a, c, b, d\}$ and $\{a, e, d\}$. They mine a Petri net, seen in figure 1.1, and the state machine seen in figure 1.2. Both models represent the same behaviour, but look different. First of all, the Petri net models b, c and c, b with a structure specific to Petri nets. This makes the Petri net *look* more simple, but is it actually easier to read and investigate? The state machine is easy to follow, but will it grow more convoluted as it models more software behaviour? Is one of these models more 'correct' than the other?

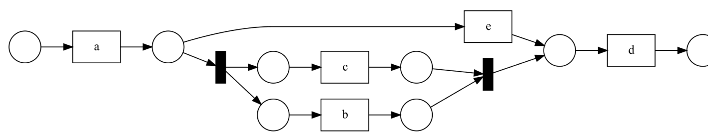


Figure 1.1: A Petri net

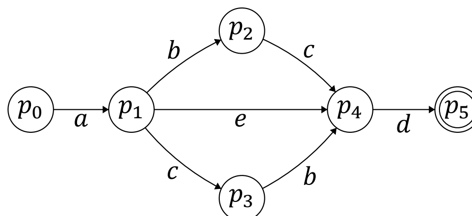


Figure 1.2: A finite state machine

In this example, the amount of observed software behaviour is small, but in real life there will be many methods and many orders they can appear in. This will impact how large these models become. Process mining tools and state machine inference tools will try to model the behaviour as efficiently as possible. This can impact the amount of behaviour they can model and how the model looks. If one were to look purely at how much behaviour could be recognised, the 'best' model may be extremely large. Additionally, it is possible that the 'worse' models only deviated slightly. Measuring quality in a deeper, more meaningful way is done differently for both of these concepts. This makes it difficult to make a fair and useful comparison.

1.1 Research Objectives

So, how would a company compare state machine and process mining tools to analyse their software and determine which one to choose? This brings us to the main goal of this thesis: to investigate how a comparison can be done and evaluate the practical use of various programs and algorithms. To achieve this goal, this work will answer the following sub questions:

- How can Petri nets and automata output be compared?
- Can the programs produce a model in feasible time and memory?
- How accurate and correct are the produced models?
- How complex are the models?
- Do data set characteristics have influence on performance?

This research is meant as an exploratory work to the connect results of process mining and state machine inference in the terms of analysing software. Specifically, this work limits the output models to 'finite state machines' and 'Petri nets', created by the tools *FlexFringe*, *MINT*, *PRINS* and *ProM*.

1.2 Thesis Structure

This thesis is organised as follows. Section 2 will introduce the reader to the relevant concepts: type of input, structure of the inferred models, inference methods and evaluation techniques. Then, section 3 will give an overview of inference performance as reported by other works. Section 4 will describe the set-up of the experiment and evaluate the choices made in this regard. The evaluation (4.4) includes an analysis of the performance metrics that were used. All results obtained through the experiment will be reported in section 5, and will be discussed thoroughly in section 6. This section will also go into the limitations of this research. Lastly, sub questions and the main research question will be answered in section 7.

2
Background

The tools that will be compared in this work, utilise different algorithms for inference and output the model in different forms. This chapter will first give a definition of the input (software execution traces) and the output (state machines and Petri nets) created. After this, the model inference techniques for each tool will be explained. Lastly, the theory of various evaluation techniques will be elaborated upon.

2.1 Software Execution Traces

A program or software execution log is a set of *traces* generated by the execution of some program. A trace is a sequence of *events*, where each event contains certain variables. To give a simple example, say a log is created in a smoothie bar. A trace would be: *take order* → *cut fruit* → *add fruit to blender* → *add milk* → *add cinnamon* → *turn on blender* → *settle bill* → *serve smoothie*. The actions in this trace are the events, which can vary in type and order in different traces. The smoothie bar log L can contain multiple traces $\sigma \in L$. A trace σ is a finite event sequence $\langle e_1, ..., e_n \rangle$, where each event usually has some parameters, such as the time, date or other variable values [53].

In software execution logs, it is common to only have *positive traces*. Positive means the behaviour in the trace is feasible in the context, i.e. the software has the ability to execute this order and type of events. A *negative trace* would be behaviour that the software cannot exhibit [53]. In the smoothie bar log, a negative trace could be to omit the '*turn on blender*' event, as this would not result in the customer getting a smoothie.

2.2 Finite State Machines

A finite state machine (FSM) is a computational model that can be represented in a state diagram, an example can be seen in figure 2.1. The circles labeled $q_0, ...q_5$ represent *states*. State q_0 is the *start state* and the double circled q_4 represents an accept state. One can move to another state with the transitions $e_1, ...e_6$. State machines are useful for recognising patterns in data [43, p. 35-36].

The formal definition of an FSM is a 5-tuple $(Q, \Sigma, q_0, F, \delta)$, where:

- Q is a finite set of *states*
- Σ is a finite set of symbols (*events*), also called the *alphabet*
- $q_0 \in Q$ is the *start state*
- $F \subseteq Q$ is the set of accept states
- $\delta : Q \times \Sigma \longrightarrow Q$ is the *transition function*

[43].

In the process mining field, a similar concept is called a *transition system*. In a transition system, the set of states can be infinite and the start and end states are not necessarily defined. In practice, most transition systems will have a finite state space and defined start/end states, and thus can be referred to as FSMs [1, p. 58].

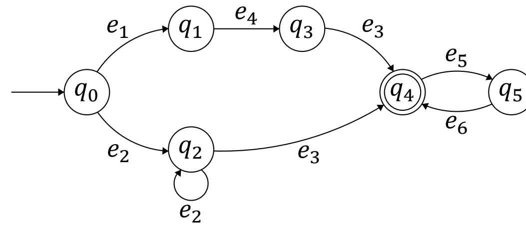


Figure 2.1: Example of a state diagram

Determinism

An FSM can be determinisic (DFA) or nondeterministic (NFA). An automaton is deterministic if:

1. Each state $q \in Q$ has exactly one outward transition for each symbol in the alphabet.
2. All transition labels e_n are in the alphabet: $e_n \in \Sigma$.

This means the example 2.1 would be an NFA. However, for clarity, all transitions leading directly to a reject state are not shown. Instead, it is assumed that this automaton rejects when it encounters symbol e_n in a state with no outgoing transition for e_n.

Every NFA can be transformed to an equivalent DFA, although the resulting DFA can have a (much) larger amount of states that the original NFA. The formal definition of an NFA differs in the transition function; rather that producing the next state, it produces the *set* of possible next states [43, p. 35-36].

Probabilistic Deterministic Finite Automata

The transition function of a DFA outputs exactly one state, whereas an NFA transition function can produce a set of states. A Probabilistic Finite Automaton (PFA) produces a weighted set of states. A deterministic PFA (PDFA) has a probability assigned to each event, and all outgoing transitions of a state sum to 1.

The formal definition of a PDFA is a 6-tuple $(Q, \Sigma, q_0, \delta, S, F)$, where:

- Q is a finite set of *states*
- Σ is the *alphabet*, a finite set of symbols (events)
- $q_0 \in Q$ is the *start state*
- $\delta : Q \times \Sigma \longrightarrow Q \cup \{0\}$ is the *transition function*
- $S : Q \times \Sigma \rightarrow [0, 1]$ is the *symbol probability function*
- $F : Q \rightarrow [0, 1]$ is the *final probability function* where $\forall_{q \in Q}\, F(q) + \sum_{e \in \Sigma} S(q, e) = 1$

This allows for easy calculation of the probability of a sequence of symbols occurring. Instead of defining F, probability functions can also be calculated over Σ^n for some PDFA $\mathcal{A}$: for all $n \geq 1$, $\sum_{s \in \Sigma^n} \mathcal{A}(s)$ where s is a sequence of observed symbols [51].

Extended Finite State Machines

The Extended Finite State Machine (EFSM) model was meant to generalize the FSM model. In this model, data registers are not part of set of states S, but instead its operations are modeled in transitions [7]. Essentially, the EFSM is an FSM with memory that holds variables. A transition is no longer enabled by encountering its symbol alone, it can also have conditions for the current value of the variables [53].

The formal definition of an EFSM is a 7-tuple $(Q, \Sigma, q_0, F, M, U, \delta)$, where:

- Q is a finite set of states
- Σ is a finite set of symbols
- $q_0 \in Q$ is the *start state*

- $F \subseteq Q$ is the set of accept states
- M is a finite set of variables in memory, assigned to values
- $U : L \times M \longrightarrow M$ is the update function
- $\delta : Q \times \Sigma \times M \longrightarrow Q$ is the *transition function*

[53].

2.3 Petri Nets

A Petri net is a modelling language that is able to express concurrency and it can be represented with a diagram [1, p. 59-63]. An example of a Petri net can be seen in figure 2.2.

The formal definition of a Petri net is a 3-tuple (P, T, F), where:

- P is a finite set of *places*
- T is a finite set of *transitions* with $P \cap T = \emptyset$
- $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation; a set of directed arcs from places to transitions and vice versa

[1, p. 59-63].

Places correspond to states of an FSM and the transition squares correspond to events. A black square is a silent transition. This does not correspond to an event in the trace, but effectively enables a skip. In figure 2.2 this means that neither b or c need to be fired after a.

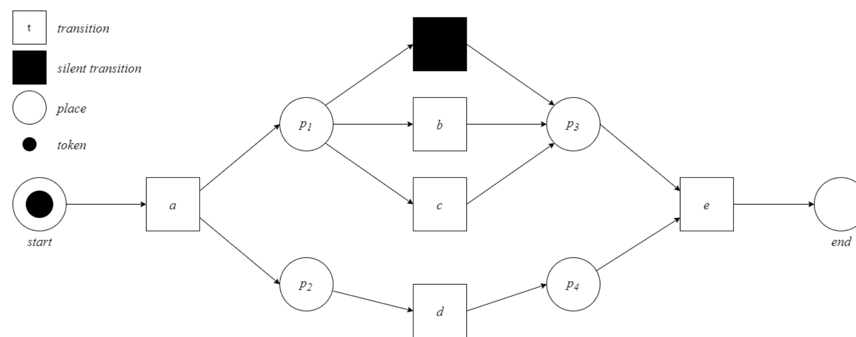


Figure 2.2: Example of a marked Petri net

The structures in a Petri net are subject to different firing rules. *Tokens* can enable transitions and flow through the net according to these rules:

- *Transition* a has multiple outbound arcs. This models *concurrency*, meaning that after a both d and b, c or the silent transition can be fired. If a is fired, the *token* in *start* is consumed and 2 tokens are produced for places p_1, p_2. Thus the beginning of a valid trace in this net can be ad (silent transition fired), adb, adc, abd, acd. This also means that this Petri net is non-deterministic as it moves to p_1 and p_2 with transition a.
- *Place* p_1 has multiple outbound arcs. This models *choice*, meaning that either b, c *or* the silent transition can be fired from p_1. enabling one of these transitions consumes the token in p_1 and produces a token for p_3.
- p_2, d, p_4 models a simple sequence. The token in p_2 is consumed to enable transition d, and a token is produced for p_4.
- *Transition* e has multiple inbound arcs, and needs a token from each place connected to these arcs to be enabled. Therefore, e can only be fired if there is a token in both p_3 and p_4.

The *marking* M of a Petri net is defined as the places that are holding a token. The initial marking of the example net is $[start]$, which enables transition a. After a fires, the marking will be $[p_1, p_2]$. A *firing sequence* σ is a sequence of transitions $\langle e_1, ..., e_n \rangle$ for which markings exists that enable them in this order, i.e. a valid sequence of events [1, p. 59-63].

Workflow Nets

A workflow net (WF-net) is a type of Petri net. It has exactly one source place i, one sink place o and all its places and transitions are on a path from the source to a sink. This concept is relevant as it is a more intuitive and logical model to define a process. In real world examples, there is usually a begin and and end to a procedure, even if the steps in between may vary [1, p. 65].

Soundness

The ultimate goal of process mining is to model a valid process. If, for example, some transition t does not have outbound arcs, dead end has been reached. The process is now stuck and no token will reach the sink place. Such a model can be classed as 'incorrect'. This is the notion of *soundness* in WF-nets. A DFA can be seen as sound by definition; since each state has one outward transition for each $e \in \Sigma$, the machine will always either reject on seeing the symbol, or continue to another state. A WF-net is sound if it satisfies the following requirements:

1. *Safeness*: a place can hold no more than one token at the same time.
2. *Proper termination*: the process ends in a sink place. At termination, only the sink place in the WF-net has a token.
$$\forall_M (i \overset{*}{\rightarrow} M \wedge M \geq o) \Rightarrow (M = o)$$
3. *Option to complete*: for every marking M, there exists a firing sequence that ends in the sink place.
$$\forall_M (i \overset{*}{\rightarrow} M) \Rightarrow (M \overset{*}{\rightarrow} o)$$
4. *No dead transitions*: for every transition e, there exists a marking that enables e.
$$\forall_{e \in T} \, \exists_{M,M'} \; i \overset{*}{\rightarrow} M \overset{e}{\rightarrow} M'$$

Requirement (3) implies (2), as the option to complete means it is indeed possible to properly terminate [46][1].

2.3.1 Conversion to Petri Net from FSM

An automaton can easily be transformed into a Petri net, as done by Habben-Jansen [18] and S. J. J. Leemans et al. [25]. This is done by creating a place p for each $q \in Q$, a new start place p_0 and an end place p_{end}. Start state q_0 will be connected to p_0 with a silent transition and all $q \in F$ will be connected to p_{end} with a silent transition. After this, a transition t is added for each transition (q, e, q'), and arcs (p_q, t) and $(t, p_{q'})$ to corresponding places are created. The definition of soundness will hold for a DFA. Each transition has only one outbound arc, as a DFA does not model concurrency and a transition goes to one state only. Thus, there will never be more than one token, resulting in a *safe* Petri net. A DFA must have an outward transition for each symbol of the alphabet, meaning it is always possible to *complete* and *terminate properly*. However, DFAs can contain dead states: these are states with only self-loops. This would violate requirement (4) [18].

2.4 Model Inference Techniques

This section will elaborate upon the inference methods used in this work. The first three methods can infer Petri nets and are included in the ProM [49]: the directly follows miner, the Hybrid Integer Linear Programming based miner and the Inductive Miner. After, the Evidence-Driven blue-fringe state-merging algorithm [23] that was implemented in Flexfringe [52] will be discussed. Lastly, MINT's [53] inference technique and PRINS' [42] technique to improve scalability will be explained.

2.4.1 Directly Follows Miner

This is a miner by S. J. J. Leemans et al. [25] that outputs a *directly follows model* (DFM), which is somewhere in between an FSM and a Petri net. Unlike an FSM, this model does not focus on the state of a process, but on the order of activities. States represent events and edges represent a 'directly follows' relation between events. DFMs do not model concurrency like Petri nets generally do and

thus are more scalable and easier to interpret. The models generated by this approach are always sound [25].

The miner works as follows. The user can set a threshold for the minimum amount of traces that will be kept for mining. The initial DFM is generated by adding nodes and edges for each trace in the log. It keeps track of how often an edge (i.e. a directly-follows relation) occurs in a log. All traces that contain the least frequent edges are removed. Then, a new DFM is generated on the filtered log and the procedure starts again. It is repeated until the threshold is reached. For example, if the threshold is 0.8, the procedure will stop before the size of the filtered log becomes smaller than 80% than that of the original log [25].

Since the DFM is modelled on traces, without removing edges from the model directly, completion and proper termination is always possible. This also ensures there are no dead transitions. There is no concept of multiple arcs coming out of an edge, so the DFM is 1-safe. Thus, the DFM is sound. S. J. J. Leemans et al. [25] convert the DFM to a Petri net as described in section 2.3.1, which means the resulting Petri net is also sound.

2.4.2 Hybrid ILP Miner

This process discovery algorithm was proposed by Zelst et al. [55]. It is a language-based region miner and builds upon the Integer Linear Programming (ILP) formulation of previous work. The idea of mining language-based regions is to use a language as input. The 'language' in this case is and event log L with a set of activities T [1, p. 218-222]. First, language-based regions will be explained. Then, the workings of the base algorithm are explained in terms of the ILP-formulation it uses and how it applies them to causal relations. Lastly, its approach to over-fitting is discussed: sequence filtering.

Language-Based Region Mining

Suppose a Petri net N is mined with a log L that has a set of events T. One could produce a Petri net with the set of transitions being T. If there are no input places for a transition, each transition is always enabled. This means the net can reproduce any trace σ in L, in fact, it can produce any trace over T. Adding a place p_i can restrict behaviour, as the transition it connects to now needs a token in p_i to be able to fire. The idea of language-based region mining it to map places to transitions, while not imposing restrictions that prevent any $\sigma \in L$ from being valid in net N [1]. In conclusion, in the context of a Petri net, a region represents place with a set of in- and output transitions.

The ILP-Formulation

An ILP problem is defined as a function that is optimised, while adhering to certain constraints for the function variables. For each potential region p, a function is minimised. A potential region is defined by a set of input transitions, a set of output transitions and an initial marking m_0 from which p is reached. This function consists of three parts. The first part denotes how many tokens are in the initial marking: c. The second part f_1, is a sum of all tokens *produced* by all input transitions of p. The third part, f_2, is the sum of all tokens can be *consumed* by the output transitions of p. Then, the function $c + f_1 - f_2 \geq 0$ is minimised. The result of this function must be ≥ 0, as a lower number would mean more tokens were consumed than the amount of tokens p had available. This definition concerns a *dual* variable region, as the in and output transitions are considered. f_2 can be omitted to obtain a *single* variable region, which does not differentiate in- and output transitions. A dual variable region is needed to model self-loops the typical Petri net patterns, such as parallelism. However, this does add more variables to the optimisation problem Zelst et al. [55].

So, what are the ILP constraints and how does this work in the miner? First of all, the algorithm tries to find potential places by mining *causal relations* between events. A causal relation between events a, b exists if any sequence in the log contains $\sigma = \{..., a, b, ...\}$. These causal relations will serve as the constraints for the ILP problem: e.g. a potential place must model a as input transition and b as output, without allowing causal relations to do not exist in the log. Then, the ILP problem is solved to obtain a place that restricts the initial disconnected set of transitions to a causal relationship as seen in the log. The ILP is solved for each causal relation [55].

The previous paragraphs explain the basic idea of this miner in a simple way. For more specifics of the optimisation function, hybrid regions and the exact ILP constraints, one is advised to read the work by Zelst et al. [55].

Sequence Encoding and Filtering

The proposed hybrid ILP miner has problems dealing with unusual traces in event logs. The algorithm will attempt include all behaviour from the log. Thus, an unusual trace can result in very restrictive ILP constraints, causing the algorithm to not find a place that does adhere to all other traces. This problem will be addressed by applying filtering [54].

A prefix set of the log is created. For example, a log $L = \{\langle e_1, e_2, e_3\rangle, \langle e_1, e_2, e_4\rangle\}$, has the prefixes $\overline{L} = \{\varepsilon, \langle e_1\rangle, \langle e_1, e_2\rangle, \langle e_1, e_2, e_3\rangle, \langle e_1, e_2, e_4\rangle\}$. The idea is to count the occurrences of all prefixes, and filter all occurrences below a certain threshold. The prefix set is used to create an acyclic graph. This graph contains all prefixes in the set of nodes and has the ε prefix as root. Nodes are connected if the source is node is a prefix for the target node. Transitions are weighed with the frequency of it occurring in the log L. The weight of a transition coming form node n with frequency f is computed as $\frac{f}{f_{n,max}}$, where $f_{n,max}$ is the highest frequency on the out transitions of the node n. If this fraction is below the the threshold, the transition is removed from the graph. The prefix nodes are called *sequence encodings*. Only causal relations present in the filtered graph will be added to ILP constraints [54].

2.4.3 Inductive Miner

The inductive miner (IM) is a state-of-the art process discovery approach, introduced by S. J. J. Leemans et al. [28]. The algorithm and its variants, such as the inductive infrequent [29] miner, guarantee a sound process and preservation of fitness [27] [1, p. 222-236]. This subsection will introduce the basic inductive miner and the ‘infrequent - all operators’ extension (IMfa).

The inductive miner does not use Petri nets internally, but *process trees*. A process tree consist of nodes with operators $\bigoplus = \{\rightarrow, \times, \wedge, \circlearrowleft\}$ and leaves with events. Figure 2.3 gives an example of a process tree with the sequence operator $\rightarrow$ as root node. This operator will execute its children in sequential order, meaning this process always starts with the redo loop $\circlearrowleft$. A redo loop has a ‘do’ and a ‘redo’ part. The leftmost child first is the ‘do’ part and is executed first. If the loop is entered, the other children are executed: the redo part. This also means the ‘do’ part is executed again. The parallel operator $\wedge$ models sub traces $\{\langle e, f\rangle, \langle f, e\rangle\}$. The sequential part of the tree ends with the choice operator $\times$, meaning the trace will end with either c or d. A process tree can be mapped to a workflow net, and this net will always be sound [1, p. 80-83].

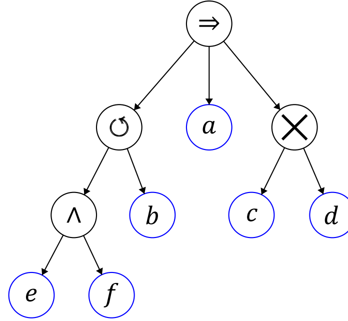


Figure 2.3: Example of process tree,
→ = sequential composition, × = exclusive choice, ∧ = parallel composition and ↺ = redo loop

The inductive miner aims to discover the process tree for log of traces. It starts with constructing a *directly follows graph* (DFG) of the log L: $G(L)$. This is a graph where each event is a state. The edges in the graph denote directly follows relations: states a, b are connected with an arrow if there exists a trace in L where $\langle ..., a, b, ...\rangle$. The start states in the graph Σ_L^{start} are the events the logs start with. The same goes for the end states: Σ_L^{end}. After constructing $G(L)$, the algorithm will try to discover *cuts* of $G(L)$ with patterns that correspond to the four operators $\bigoplus = \{\rightarrow, \times, \wedge, \circlearrowleft\}$ [28][27][1].

If a cut is found, log L is split into sub-logs. Cut discovery continues recursively: DFGs for the split logs are created, the search for a cut starts on these sub-graphs and the sub-logs are split. The logs are split differently depending on which type of cut was found. This will be illustrated with an example log $L = \{\langle a, b, c\rangle, \langle a, c, b\rangle, \langle a, d, e\rangle, \langle a, d, e, f, d, e\rangle\}$ by S. J. J. Leemans et al. [28]:

- Sequence → split: effectively separates the sequenced activity sets. The new log L_2 becomes $\{\langle b,c\rangle,\langle c,b\rangle,\langle d,e\rangle,\langle d,e,f,d,e\rangle\}$. It is not necessary to recurse on log $L_1=\{a\}$.
- Exclusive choice × split: in l_2, an exclusive choice cut can be found at $\{b,c\},\{d,e,f\}$ as they have no events in common. The log is split into sets with common events: $L_3=\{\langle b,c\rangle,\langle c,b\rangle\}$ and $L_4=\{\langle d,e\rangle,\langle d,e,f,d,e\rangle\}$.
- Parallel ∧ split: log L_3 contains a parallel cut. The split logs consist of each parallel event set: $\{b\}$ and $\{c\}$.
- Redo loop ↺ split: log L_4 has a repetitive sequence d,e. The loop cut is $\{d,e\},\{f\}$ and is split on the 'do' and the 'redo' part: $L_5=\{\langle d,e\rangle\}$ and $L_6=\{\langle f\rangle\}$

[28][27][1].

The recursion ends by matching with a base case. There are two base cases, which are tried in order. The first base case that is checked is `emptyLog`, which is returned when the log is empty. It returns the silent event τ. The second case is `singleActivity`. It applies when all traces in the log contain only one type of event. This single event is returned as a leaf [27].

It is possible that no base case applies or no cuts can be found. To avoid not returning a process tree, the IM algorithm has a fall-through function, which tries to apply the following patterns in order:

1. `emptyTraces`: if an empty trace is in the log, $\epsilon\in L$, an exclusive choice construct with a silent event as child is added: $\times(\tau,..)$. The recursion is now able to continue on sub log without empty traces.
2. `activityOncePerTrace`: if some event e occurs exactly once in each trace of the log, event e is filtered from L resulting in L'. The tree structure $\wedge(e,IM(L'))$ is added.
3. `activityConcurrent`: this fall-through filters a specific activity e from L, thus obtaining L' where e is filtered from all traces and L'' which contains all traces with everything except e filtered. If a cut is found for L', event e is added parallel to $IM(L')$.
4. `strictTauLoop`: if a loop is detected, this fall-through applies. To detect the loop, the log splits traces where an end event is followed by a start event. If at least one split was found, the structure ↺ $(IM(L'),\tau)$ is added.
5. `TauLoop`: similar to the previous fall-through, except the traces are split on every occurence of a start activity. If at least one split is found, structure ↺ $(IM(L'),\tau)$ is added.
6. `flowerModel`: this fall-though requires a log without empty traces. It returns a model that allows any behaviour over the event set $\Sigma(L)$ of L, resulting in the following tree: ↺ $(\times(e\in\Sigma(L)),\tau)$

Thus, returning the flower model is the absolute last resort [27].

This algorithm preserves precision, i.e. the model does not allow extra behaviour, for the choice, sequence and parallel operators. The redo operator introduces unbounded behaviour, which cannot exist in a log with finite traces and thus is not precision preserving. IM also guarantees soundness, as process trees are sound by design, and perfect fitness [27].

Inductive Miner - All Operators

The 'all operators' (IMa) extension adds three operators: silent event τ, interleaved ↔ and inclusive choice ∨. The interleaved operator executes all of its children, without overlap. This means a child needs to finish its execution before the another child can start execution. The silent event models *optionality*. If a silent event is in the process tree, a new state can be moved to without firing an event. Silent events are only relevant when they are a child of a redo or an exclusive chose operators. For the inclusive choice, *at least one* of its children is executed. The followed traces of children can overlap if multiple are executed. The following subsection will elaborate upon the detection and log splitting for the interleaved ↔ and inclusive choice ∨ operators, as the silent event τ is handled by a fall-through [27].

The log splitting functions for $\{\rightarrow,\times,\wedge,$↺$\}$ of IMa are the same as those of the base IM. New functions for the additional operators are added, and illustrated with example:

- Interleaved ↔ split: the log is split by the corresponding subtraces. For example, an interleaved cut is found in log $L = \{\langle a,b,c\rangle, \langle b,c,a\rangle\}$. This will split the log into $L_1 = \{\langle b,c\rangle^2\}$ and $L_2 = \{\langle a\rangle^2\}$.
- Inclusive choice ∨ split: works like the interleaved split, but assumes that there are no empty traces in L, courtesy of the `emptyTraces` fall-through, to preserve fitness. If an inclusive choice cut is found in log $L = \{\langle b,c,a\rangle, \langle a\rangle, \langle b,c\rangle\}$, it is split into $L_1 = \{\langle b,c\rangle^2\}$ and $L_2 = \{\langle a\rangle^2\}$.

[27]

The base cases and fall-through cases are the same as those of the basic IM and do not need to be extended. The `emptyTraces` case detects the silent event τ pattern.

Inductive Miner - Infrequent - All Operators

Both the base IM and IMa have issues correctly modelling infrequent and deviating behaviour in logs. If a log contains a deviating trace, that occurs only once, it will create a process tree that models this trace as well. Such a tree may be many times larger or introduce extra behaviour. If this deviating trace is only one out of hundreds, it may not be desirable to output an overly complicated model to obtain a replay for the deviating trace, especially if it lowers the precision a lot. This is addressed with filtering at every step of the IM algorithm, using a deviation threshold f [27].

Cut detection starts the same as in the base IM. However, if the cut detection does not succeed, the DFG is filtered based on f and the search for a cut starts with the filtered DFG. If a cut is found, the log is split and recursion continues. The log splitting functions are also slightly changed. When entering the log splitting function for the chosen operator, infrequent log behaviour that violates this operator is filtered out. However, parallel and inclusive choice operators allow for any behaviour and thus nothing will be filtered. The functions split logs as in the IM algorithm when there are no deviating events [27][29].

The `singleActvitiy` base case and `emptyTraces` are also slightly altered. A more detailed explanation of this and the whole `IM` algorithm and all of its guarantees, can be found in the PhD thesis of Leemans [27]. IMfa does not preserve fitness, but it does preserve soundness.

2.4.4 Evidence-Driven Blue-Fringe State-Merging

FlexFringe implements the Evidence Driven blue-fringe State Merging (EDSM) [23] algorithm. The idea of state merging is to start with a tree, the *prefix tree acceptor*, that accepts all traces in a log and start merging compatible states. The decision whether states should be merged is based on an evaluation function, such as Akaike's Information Criterion (AIC).

Blue-fringe EDSM

The prefix tree acceptor is constructed by iterating through a log of traces. From a start node, transitions are added for each unique first event in each trace. Then, the second events are added as transitions to their proper prefix. This is done for all events in all traces in the log [52]. An example can be seen in figure 2.4.

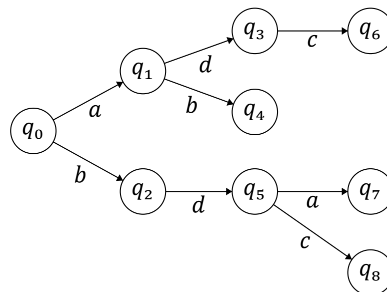


Figure 2.4: Prefix tree acceptor for log $L = \langle ad, ab, adc, bda, bdc\rangle$

The basic concept of EDSM is to iterate over all node pairs and compute a merge score. The highest valid merge is performed. The first step is to compute whether nodes are equivalent, i.e. if their in- and output transitions are the same. If they are, the potential merge will be scored.

When some pair (q_1, q_2) with the highest score is found, they are merged as follows: all in- and out transitions of state q_1 are transferred to q_0. If both states were accept states, the merged state is also accepting. This is the same if one state is a reject state. An accept state and a reject state cannot be merged [23]. In FlexFringe, if a merge causes non-determinism, the target states of the non-deterministic transitions are also merged [51].

The aforementioned algorithm will merge nodes in a random order and has many candidate pairs. FlexFringe implements a more restrictive variant: the blue-fringe algorithm. This algorithm limits the choice of candidate nodes, by colouring nodes and only considering blue/red pairs. In the beginning, the start state is coloured red and its children are coloured blue. Throughout the state merging, the following characteristics will remain constant:

- Red nodes form a connected graph
- If a red node has children that are not red, they must be blue
- Blue nodes are not the endpoint of any transition, i.e. they are roots of isolated trees

Instead of iterating over all pairs, only the merge score of red/blue pairs will be computed. If one of the merges is consistent, the merge with the highest score is performed. Merge candidates scoring below a certain score, are coloured red. The children of the new red states are coloured blue. This restriction speeds up the state merging significantly [23][51].

Akaike's Information Criterion

FlexFringe's implementation checks *consistency* rather than equivalence of a merge candidate. This is can be done with various algorithms such as Alergia, the likelihoodratio and AIC. If a merge is performed on some PDFA $\mathcal{A}$, it becomes the more restricted PDFA $\mathcal{A}'$. Akaike's Information Criterion computes the difference in transitions and the loglikelihood between $\mathcal{A}$ and $\mathcal{A}'$. These two differences are then deducted. The AIC is computed as follows:

$$2(|\mathcal{A}| - |\mathcal{A}'|) - 2\left(\sum_{q \in Q, a \in \Sigma} C(q,a)\log(S(q,a)) - \sum_{q' \in Q', a \in \Sigma} C(q',a)\log(S'(q',a))\right) > 0,$$

where $|\mathcal{A}|$ is the number of transitions in $\mathcal{A}$, $C(q,a)$ is the frequency count of an event a in state q and $S(q,a)$ is the symbol probability function of an event a state q. Thus, a merge decreasing the log likelihood and/or the number of transitions in $\mathcal{A}'$ compared to $\mathcal{A}$, minimizes the AIC. Such a merge is consistent [51].

2.4.5 MINT

MINT is a generalised Model Inference Technique by Walkinshaw et al. [53] to infer EFSMs. However, the update functions of the EFSMs are not inferred, which means the resulting models are not complete EFSMs and only capture whether sequences of events and variable values are possible. This technique builds upon existing state-merging approaches such as EDSM [23] and GK-tails [32]. In contrast to these methods, MINT is a modular algorithm, allowing insertion of various data *classifier inference techniques*. These are type pattern recognition techniques, or more specifically, classifiers. Classifiers identify patterns in order to assign an output class to a set of observations. This subsection will explain MINT's inference technique and the classifier inference technique `AdaBoost` [53].

Inference Algorithm

This algorithm starts with the inference of a set of classifiers. The classifiers correspond to the event names in the traces and are meant to predict the next event in a trace. So, there is a classifier for each unique event in the log. The rest of the inference algorithm is much like the the aforementioned EDSM: A prefix tree acceptor (PTA) is generated and pairs of states are merged, in a way that ensures determinism. However, there are a few key differences in the PTA and extra constraints for merging pairs.

during preprocessing, a classifier is learned for each event in the log. E.g. a classifier for an event a can receive an event b as input and will 1 if b is likely to follow a, 0 otherwise. Unlike in the

basic state merging algorithm, the PTA transitions include a set of data variables in addition to the event label. Another difference in the PTA is that pairs of states only share a prefix if the classifiers produce the exact same result both states.

The classifiers are also incorporated into the state merging algorithm. The inference algorithm iterates over a set of chosen candidate state pairs and after each merge, the model is checked for consistency with the classifiers. If a model $\mathcal{A}'$ generated by a merge in model $\mathcal{A}$ is not consistent with the classifiers after a merge, $\mathcal{A}'$ is discarded and the node pair is stored in memory as a failed merge. Iteration continues with $\mathcal{A}$.

The candidate pairs are not random pairs, they are chosen by considering several variables and calculating a merge score. The score is computed by iterating over pairs of *equivalent transitions* of a pair of states. In each iteration, the score is incremented by 1 and added to the merge score of the target states of the transitions. Transitions are equivalent if their data variables are considered equivalent by its classifier. This function also includes an optional minimum merge score k. Any pair of states below that score is never considered equivalent.

AdaBoost

Adaptive Boosting (AdaBoost) is an algorithm by Freund and Schapire [16] that aims to find classification rules. The key idea of *boosting* is to combine prediction rules (*hypotheses*) that are not very accurate to find an accurate prediction rule. The algorithm receives training samples (x_i, y_i), where x_i is the input and y_i its label. In this context, that means the event name e_i and its successor e_{i+1} respectively. In short, this method will start with a *weight vector* that has equal weights for all data points and use it as input for a *weak learner* to train a model. A weak learner creates predictions with variable accuracy, but at least slightly better than random guessing. The samples that are wrongly classified by the weak learner receive a higher weight. This new weight vector is used in a subsequent iteration, which trains another model and updates the weight vector the same way. This continues until a satisfactory model is created or for a specific number of iterations [16].

More specifically, the algorithm with N training samples works as follows:

- A weight vector w^1 of length N is initialized with an uniform distribution. All weights are $\frac{1}{N}$.
- A specific number T is set to determine the amount of iterations
- Iterate T times, $t = 1, 2, .., T$:
  1. Compute distribution:
     $p^t = \frac{w^t}{\sum_{i-1}^{N} w_i^t}$.
  2. Train a model with a weak learner and p_t, this outputs a weak hypothesis h_t.
  3. Compute the error ε_t of the weak hypothesis over all training samples:
     $\varepsilon_t = \sum_{i-1}^{N} p_i^t * |h_i(x_i) - y_i|$.
  4. Compute parameter β as:
     $\beta_t = \frac{\varepsilon_t}{1-\varepsilon}$.
  5. Update weight vector by multiplying each weight i by $\beta_t^{1-|h_i(x_i)-y_i|}$:
     $w^{t+1} = w_i^t * \beta_t^{1-|h_i(x_i)-y_i|}$.
- Output final hypothesis h_f:

$$h_f(x) = \begin{cases} 1 & \text{if } \sum_{t=1}^{T} \left(\log \frac{1}{\beta_t}\right) h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \log \frac{1}{\beta_t} \\ 0 & \text{otherwise} \end{cases}$$

[16]. In the context of MINT, a classifier is made for each unique event during pre-processing. This algorithm is run for each event e_i in the log and the $[0, 1]$ prediction outputs correspond to 'e_i does not follow this event' and 'e_i follows this event' respectively. During state merging, the in- and outgoing events of a proposed merged state are checked for consistency with these predictions.

2.4.6 PRINS

PRINS is a technique by Shin et al. [42] that aims to address scalability problems of inference algorithms. The idea is to use an inference tool that produces deterministic models, such as MINT, and infer a model for each *component* in a log. An example of a component in a software log could be a class or a module. After this, all component models are merged into one model. This model is determinised with a novel *hybrid determinisation* technique to produce the final model.

The models are inferred on the component logs. A component log consists of traces with events called by the component. For example, a log with events e_c, where c is the component could be: $\{\langle a_1, b_1, c_2, d_1\rangle, \langle a_2, b_1, c_2, d_1\rangle\}$. The two component logs will then become: $L_1 = \{\langle a_1, b_1, d_1\rangle, \langle b_1, d_1\rangle\}$ and $L_2 = \{\langle c_2\rangle, \langle a_2, c_2\rangle\}$. An inference tool is then used to obtain the set component models $M_c = m_1, m_2$.

Combining Component FSMs

After the set of component models is obtained, PRINS initiates the *stitching* stage. The idea is that all models are combined according to the sequence of events in the traces of the log. This stage consists of four phases: *partition*, *slice*, *append* and *union*. The idea is to loop through all traces in the data set and create a sub model for each trace. Afterwards, the sub models are merged to one model in the *union* phase.

Partition. The loop for one trace starts by initialising an empty model m_{sub}. Then, it partitions the trace into sequences created by one component. So, for the first trace in the example, the partition P becomes: $P = \langle l_{c,1}, l_{c,2}, l_{c,3}\rangle$ with $l_{c,1} = \langle a_1, b_1\rangle$, $l_{c,2} = \langle c_2\rangle$ and $l_{c,3} = \langle d_1\rangle$. Then, the slice and append phase will be initiated *for each* of the sequences l_c in this partition [42].

Slice. For each sequence l_c, the model m_c of the corresponding component c will be sliced and appended to the m_{sub} for the current trace. Model m_c is sliced to obtain a sliced model $m_{s,c}$ that only accepts sequence l_c. This is done by reading l_c in order and traversing it in m_c. All successfully traversed states are added to $m_{s,c}$ [42].

Append. After the slice, model $m_{s,c}$ is appended to the sub model m_{sub}. If m_{sub} is empty, it is simply updated to be $m_{s,c}$. If this is not the case, the final state of m_{sub} is merged with the initial state of $m_{s,c}$. Merging the states is done by adding all incoming edges of the m_{sub} final state and all outgoing edges of the $m_{s,c}$ initial state [42].

Union. Finally, the union phase merges all initial states of every m_{sub}. The resulting model m_u accepts the same traces as all m_{sub}. This phase, as well as the append phase, can make a model non-deterministic. At the end of this step, there is a tree-like model with a branch for each trace in the log [42].

Hybrid Determinisation

Standard algorithms used for determinisation can have an exponential worst-case complexity. Other techniques can over-generalise the model, i.e. causing the determinised model to accept more behaviour then its non-deterministic source. To combat this, Shin et al. [42] propose *Hybrid Determinisation with parameter u* (HD_u). This approach merges target states of non-deterministic transition, *only if it has been merged less than u times before*. The idea is to avoid merging states many times and limiting the introduction of new behaviour in the determinised model. The approach is based on the standard algorithm *powerset construction* and *heuristics-based determinisation* [11]. The concept of heuristics-based determinisation is to recursively merge target states of non deterministic transitions. A choice of $u = 0$ results in the use of the powerset construction, and a choice of $u = \infty$ in heuristics-based determinisation [42].

2.5 Evaluation Techniques

Models are often evaluated in terms of how a model reacts to a set of test traces. These traces can be *positive* or *negative*. A positive trace is a trace that should be compliant with the model. A negative trace should be rejected by the model. A positive trace that is correctly classified is a *true positive* (TP), and one that is incorrectly classified as negative is a *false negative* (FN). Traces that are correctly classified as negative are *true negatives* (TN) and if a negative trace is classified as positive, it is a *false positive* (FP). These counts can be used to calculate the widely used metrics: recall, precision, accuracy and specificity [42][14][53][13].

This section will discuss various other evaluation techniques for models as well as Petri nets. Firstly, the F-score and the *balanced accuracy* will be discussed, as can be computed with the aforementioned metrics. After, the trace fitness will be explained. Fitness expresses how much of a positive trace is can be replayed on a model, i.e. how well a trace fits the model. For Petri-nets, this can be calculated in two different ways: token-based and alignment based. Lastly, perplexity will be discussed. This evaluates how surprised, i.e. 'perplexed', a model is by a trace.

2.5.1 F-score

The F-score is defined as the harmonic average of precision and recall. Precision p and recall r are calculated as follows:

$$p = \frac{TP}{TP + FP} \qquad r = \frac{TP}{TP + FN}$$

The F-score can be weighted with β to assign more importance to either precision or recall. The standard F_1-score is the harmonic average and is calculated when $\beta = 1$. A $\beta < 1$ assigns more importance to precision, while a $\beta > 1$ gives more weight to recall. The weighted F_β-score is computed as follows:

$$F_\beta = (1 - \beta^2)\frac{p * r}{r + \beta^2 p}$$

[17].

2.5.2 Balanced Accuracy

Accuracy expresses how many traces in the test data were correctly classified by the model. It is defined as:

$$Acc = \frac{TP + TN}{TP + FN + TN + FP}$$

Balanced accuracy BA is the mean of recall and specificity. Specificity is the true negative rate: $sp = \frac{TN}{TN+FP}$. The BA is simply calculated as:

$$BA = \frac{r + sp}{2}$$

This measures the average of how well a model can identify positive and negative traces [53].

2.5.3 Fitness

One way to compute fitness is to simply check which portion of a trace can be replayed on a model. However, this may result in a low fitness or even a fitness of 0, even if a trace *almost* fits. This can be a problem in Petri nets especially, as the transition necessary may present, but not enabled.

Token-Based

The general idea of token-based replay (TBR) fitness is to continue replaying a trace even if a transition is not enabled. During the trace replay, the produced tokens p, consumed tokens c, missing tokens m and remaining tokens r are counted:

- A *produced* token is counted each time a transition is enabled and produced a token.
- A *consumed* token is counted after a transition was enabled and produced its tokens.
- A *missing* token is counted when a transition needs to fire, but its input place contains too few tokens or none at all. E.g. a transition can have multiple inbound arcs, and needs tokens in all places connected to these arcs. If x amount of these places do not contain a token, m is incremented by x.
- A *remaining* token is counted when a token gets 'stuck', i.e. it is not consumed to fire a transition. This can be computed at the end from the other counts: $r = p + m - c$

The token-based fitness of trace σ on Petri net N is then defined as:

$$fitness_{TBR}(\sigma, N) = \frac{1}{2}\left(1 - \frac{m}{c}\right) + \frac{1}{2}\left(1 - \frac{r}{p}\right)$$

The value lies between 0 and 1, where 1 denotes a perfectly fitting trace. During replay m will always be lower than or equal to c and $c \leq p + m$ [1, p. 246-255][2].

Alignment-Based

Alignment-based fitness builds on *alignments* of traces. The idea is to align the moves done in the trace with what is possible in the model. This means that if the end of a trace is completely valid, this will positively affect the fitness. An alignment is a sequence of moves that aligns a trace to a model. An example in table 2.1 for trace $\sigma = \langle a, x, d, e \rangle$ on a straightforward model that accepts only $\sigma = \langle a, b, c, d, e \rangle$. In this alignment, 2 moves $\gg$ were needed to align the trace to the log. Additionally,

a	x	$\gg$	$\gg$	d	e
a	$\gg$	b	c	d	e

Table 2.1: Optimal alignment

the model needs an x move as it does not have this transition in the model. An *optimal alignment* is the best alignment match between a log and a model, i.e. an alignment with the lowest *cost*. Selecting the appropriate alignment is done by assigning costs to asynchronous moves and possibly silent transitions. Silent transitions can be minimised as they can make the alignment longer than it needs to be. If this is done, its cost is set much lower than that of an asynchronous move as it is technically a valid move. There can be more than one optimal alignment for a given trace and model [1, p. 256-263].

These alignment costs are used to compute a fitness value between 0 and 1, where 1 denotes a perfectly fitting trace. This is done by getting a worst case alignment $\lambda_{worst}(\sigma)$, with no valid moves, seen in 2.2 In the optimal alignment, 3 moves were necessary to align the log. In the worst case, 9

a	x	d	e	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$
$\gg$	$\gg$	$\gg$	$\gg$	a	b	c	d	e

Table 2.2: Worst case alignment

moves are necessary. The fitness is now calculated as:

$$fitness_{AL}(\sigma, N) = 1 - \frac{\delta(\lambda_{opt}(\sigma))}{\delta(\lambda_{worst}(\sigma))} = 1 - \frac{3}{9} \approx 0.66$$

[1, p. 256-263].

2.5.4 Perplexity

Perplexity is a metric expressing the probability of words [21]. The perplexity PP of some log L on a model $\mathcal{A}$ is calculated as follows:

$$PP = 2^{-\frac{1}{N}\sum_{\sigma \in L} \log_2(P_{\mathcal{A}}(\sigma))},$$

where $\sigma \in L$ are all traces in the log and N is the amount of symbols of the traces that are recognised by the model. $P_{\mathcal{A}}(\sigma)$ is the probability of a trace σ in model $\mathcal{A}$. The probability of a trace is computed by multiplying all event probabilities.

The probability of some event e_i occurring is $\frac{c_i}{|\mathcal{A}|}$, where $|\mathcal{A}|$ is the total number of transitions going out of a state. An example of this can be seen in figure 2.5, in this model, all probabilities are uniform. All probabilities coming out of a state must sum to 1. A smaller PP value indicates the model less surprised by the traces, and thus a better model than one with a higher PP on the same log.

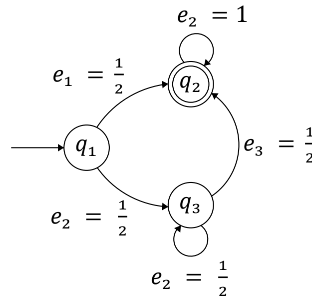


Figure 2.5: Example of a PDFA with its probabilities

2.5.5 Complexity

The complexity of a model can reflect two aspects of a model. Firstly, there is the visual aspect. A highly complex model will visually look like spaghetti; some models have so many transitions to states, it becomes impossible to discern which transitions go where. The second aspect relates to how easy it is to follow the *control flow*, i.e. the order of events, in a model.

Cyclomatic Complexity

McCabe's Cyclomatic Complexity (CC) [34] is graph-theoretic complexity metric that counts the paths that can be taken through a model. The CC of a graph G with n nodes and e edges and p connected components is defined as:

$$CC(G) = e - n + 2 * p.$$

An FSM can be represented by a directed graph. If this were to be translated one-to-one to a Petri net, it would result in:

$$CC(Petri) = |F| - (|P| + |T|) + 2 * p,$$

where $|F|$ is the number of arcs, $|P|$ the number of places and $|T|$ the number of transitions. This gives an indication of the visual aspect similar to that of a graph. However, it does not give an indication of how complex the model is. The concurrency in the Petri net introduces many more paths than the $CC(Petri)$ indicates. It would be possible to replicate the CC by constructing the reachability graph of the Petri net. A reachability graph of a Petri net has all possible markings as states, with transitions between them if it is possible to move to a certain marking. Calculating this can take an extremely long time [24].

Control Flow Complexity

A complexity metric for process models was presented by Cardoso [6]. The complexity of a process model is measured in terms of the XOR-, OR- and AND-split structures. The splits are calculated as follows:

$$\begin{aligned} CFC_{XOR}(a) &= \text{fan-out}(a) \\ CFC_{OR}(a) &= 2^{\text{fan-out}(a)} - 1 \\ CFC_{AND}(a) &= 1, \end{aligned}$$

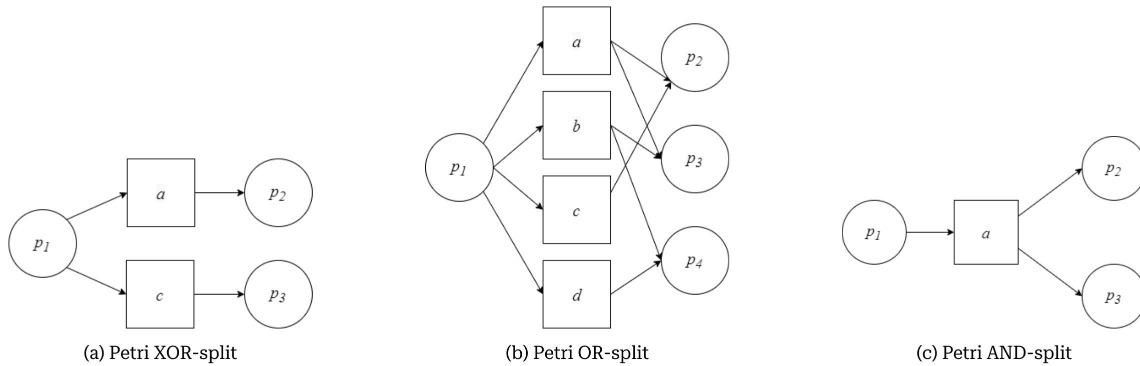
where the fan-out is the out-degree of an activity. The idea is to penalize the splits with how many extra paths it introduces. Then the absolute CFC is calculated by summing all of these splits:

$$CFC_{abs}(Petri) = \left(\sum_{i \in XOR-splits} CFC_{XOR-split} i \right) + \left(\sum_{j \in OR-splits} CFC_{OR-split} j \right) + \left(\sum_{k \in AND-splits} CFC_{AND-split} k \right)$$

Lastly, the relative CFC is calculated as follows [6]:

$$CFC_{rel}(Petri) = \frac{CFC_{abs}(Petri)}{|\{\text{XOR-splits of p}\} \cup \{\text{OR-splits of p}\} \cup \{\text{AND-splits of p}\}|}.$$

The CFC was further defined for Petri nets by Lassen and van der Aalst [24], resulting in the *extended* CFC. An example of each of the splits in a Petri net can be seen in figure 2.6. The definition for XOR and AND splits is the same, but the OR split equals the amount of the unique place sets each transition can move to. In this example, $eCFC_{XOR} = 2$, $eCFC_{AND} = 1$ and $eCFC_{OR} = |\{\{p_2\}, \{p_4\}, \{p_2, p_3\}, \{p_3, p_4\}\}| = 4$



(a) Petri XOR-split (b) Petri OR-split (c) Petri AND-split

Figure 2.6: The three different splits for Petri nets.

The eCFC can easily be computed for FSMs as well. An FSM does not model concurrency and thus only has XOR-spits. These are defined as the fan-out of a place, which is the equivalent of the out-degree of a state. All out-degrees of a state will equal the number of edges $|E|$. The number of XOR-splits will be equal to the number of states $|Q|$:

$$eCFC_{rel}(FSM) = \frac{\sum_{q \in Q} |\text{out edges } q|}{|Q|} = \frac{|E|}{|Q|}.$$

3
Related Work

To the best of the author's knowledge, no extensive work was done in terms of the direct comparison of Petri nets and state machines. However, the performance of some the inference techniques, which were explained in the previous chapter, was evaluated by their respective researchers. The performance evaluation of ProM miners, FlexFringe, MINT and PRINS will be reported in this chapter.

3.1 ProM miners

This section discusses performance of ProM miners. The Directly Follows miner did not have a performance analysis in its introductory paper, so the results of the IEEE[1] Task Force on Process Mining Process Discovery Contest (PDC) 2021 [38] are included as well.

3.1.1 Process Discovery Contest 2021

This contest included the Hybrid-ILP miner, Inductive Miner - frequent & all operators and the Directly Followers miner. It reports the F-score, the recall $= \frac{TP}{TP+FN}$ and the specificity $= \frac{TN}{TN+FP}$, which are called the positive and negative accuracy respectively. The Directly Follows miner reported a 89% F-score on average, with both a high positive and negative accuracy. The Inductive Miner - frequent & all operators has an F-score of 42.3% and a lower positive than negative accuracy. Hybrid-ILP has a similar performance, but a slightly higher F-score of 46.4%. The basic Inductive miner was included in the contest of the previous year. The PDC 2020 reported an F-score of 40%, a low positive accuracy and a high negative accuracy [38][39].

3.1.2 Hybrid ILP miner

This miner [55] and its filtering method [54] were tested by its authors, Zelst et al. [55]. The miner was tested in terms of run time performance of the single, dual and hybrid variable-based ILP formulations. The performance of the sequence filtering was testing using precision $(= \frac{TP}{TP+FP})$.

Run Time Base Miner

The hybrid ILP-miner was evaluated with regard to its run time versus the run time of a single or dual variable-based ILP formulation. The miner was run with all three settings on 10 generated logs with a number of events ranging from 2 to 12. Each log contained 5000 traces. All 3 settings were run 10 times on every log [55].

The run times for all formulations increases as the number of events increase. The hybrid ILP formulation is in between the single and dual setting for all logs. As the number of events increase, the run time of the hybrid formulation approaches that of the dual formulation formulation. The differences between all formulations are small. The authors found that using ILPs within a more reasonable time may require more research [55].

[1]Institute of Electrical and Electronics Engineers

Sequence Filtering

The sequence filtering was evaluated using artificial logs. The aim was to test performance with an increasing percentage of unusual behaviour in the log. There were 2 ground truth logs without exceptional behaviour. From these, new logs were generated with 5%, 10%, 20% and 50% noisy traces. The new noisy traces were generated by removing the head or tail from the trace, a random part of the trace body or swapping two random events in the trace. Models were inferred from the logs, using the hybrid ILP miner without filters and with filtering, using $c_c = 0.75$, $c_c = 0.5$ and $c_c = 0.25$. The initial precision values for the ground truth logs were 0.7 and 0.6 [54].

They found that for the miner without filtering, the precision dropped to 0.1 for both logs as soon as any noise was introduced, and stayed there for all extra noise added. Precision for $c_c = 0.75$ and $c_c = 0.5$ were similar too each other. The precision both drops with approximately 0.2 for 5% and rises slightly for the 10% noise log. Both decrease further as the noise percentage rises. $c_c = 0.25$ outperformed the others. While initial drop for the 5% log is similar to the other filtered miners, it had a significantly higher precision for both 50% logs. The best setting $c_c = 0.25$ still sees a decline in precision: -0.1 on one set at 50%, -0.3 on the other [54].

3.1.3 Inductive Miners

The Inductive Miners were evaluated in the PhD dissertation of Leemans [27]. This evaluation was conducted on the Inductive Miners as implemented in ProM 6.6 and compared to performance of several ProM Miners and other stand-alone miners. It was not compared to the Hybrid-ILP or the Directly Follows miner, however it was compared to the regular ILP miner. This section will elaborate upon the evaluation concerning the scalability of the implementation and their quality analysis.

Scalability

An experiment was conducted where all miners were run on a randomly generated log, which had the number of events and traces was increased synchronously for 10 rounds. Each round r, the number of events was increased by 2^r and the number of traces by 4^r. The researchers increased these numbers asynchronously in another work [26], which lead to similar results. The miners were run with 2GB RAM, and the number of successful runs were measured.

The base Inductive Miner and its ‘all operators’, ‘frequent’ and ‘frequent - all operators’ variants start to fail some runs after round 7 (128 events, 16384 traces). This is due to running out of memory. In round 8, the ‘frequent’ variant is a little better than the ‘all operators’ variant and their combination variant is a little worse than both. They perform the same in round 9 and 10: round 9 fails almost all runs and round 10 fails all runs. Most other miners started to fail runs at round 5 or 6 [27].

Model Quality

To measure fitness and precision, the miners were run on 5 real-life event logs. The logs were split into 3 parts, $\frac{2}{3}$ of the log formed a training set and the remaining part was the test set. Then, the fitness was determined with the test log and precision was measured on the full log.

The Inductive Miners all report fitness values of 1 or extremely close to 1. The lowest fitness value of an Inductive Miner was that of the ‘frequent - all operators’ variant: 0.96. Many other miners also achieved (near) perfect fitness on all sets. The ILP miner did not generate any sound models, thus its fitness and precision were not reported. The precision was much lower. The base Inductive Miner reports a precision of around 0.65 across all data sets. The ‘all operators’ variant reports the exact same precision as the base miner. The ‘frequent’ and ‘frequent - all operators’ variants report almost identical specificity values, which are higher than the other two [27].

3.2 FlexFringe

FlexFringe [51] implements several evaluation functions (Alergia, likelihoodratio, MDI and AIC) and tested all of them by inferring models on the PAutomaC learning competition [50] data sets. The sets were generated by a ground truth model with probabilities. The perplexity was computed with the real probabilities and the probabilities in the inferred models. Only the results of the PDFA problems are reported in this section.

For the PAutomac problems, AIC either reported a lower perplexity score or a score that was almost the same as the other evaluation functions. The MDI function performed the worst overall,

although there was never a large gap. Alergia and the Loglikelihoodratio often report the exact same perplexities as AIC. Some report higher perplexities, but the difference is generally small. In addition to the PAutomaC problems, an analysis was done on a labeled HDFS data set, using the AIC evaluation function. Using a test set on the model, they found it gave a low amount of false negatives and a large amount of false positives. The F-score was 0.89 [51].

3.3 MINT

Walkinshaw et al. [53] analysed the accuracy and the scalability of MINT models with four different classifier techniques: AdaBoost ([16]), C4.5 ([40], called J48), Naive-Bayes ([36]) and JRIP (Java implementation of RIPPER [9]). Models were inferred for 5 different minimum merge scores, $k = \{0, 1, 2, 3, 4\}$, on 5 different data sets. They used 5-folds cross validation and measured recall, specificity, balanced accuracy and inference time. The number of states and transitions were also reported.

Performance of the different classifiers on the balanced accuracy varied per set. There was no classifier that outperformed all others on every set. However, AdaBoost was consistently in the the top three. Performance did not only vary due to the classifier, but also the choice of minimum merge score k. Again, there was no k that performed well on all sets. For example, $k = 0$ results in (near) perfect recall with low specificity on 4 sets for all classifiers. But on one of the sets, it achieved high specificity with a mediocre recall. Some data sets performed excellently for low k values, whereas others needed a higher k (3,4) to perform well.

The inference time was the worst for the Naive Bayes classifier. It did not complete any inference for one data set, though it should be noted that the other three classifiers also did not return models for higher k values on this set. For thee of the sets, inference time got larger as k got larger. AdaBoost, J48 and JRIP performed similarly.

The amount of states and transitions in the inferred models also varied per k value. Overall, a larger k results in a larger model as less states are merged due to the higher minimum merge score. For most data sets, $k = 1$ and $k = 2$ produced models of similar size. $k = 4$ increases the size greatly for 3 of the 5 sets. This work concludes that MINT is capable of returning a good model, but there is no single combination of classifier and k that always performs well [53].

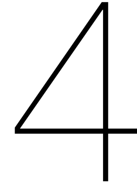
3.4 PRINS

Shin et al. [42] compare the models inferred by their algorithm, using MINT (AdaBoost, $k = 2$) for component models internally, with models that were fully inferred with this MINT configuration. In addition to this, they test their hybrid determinisation HD_u with different values for u. They use 10-folds cross validation and compare recall, specificity, balanced accuracy, inference time and model size for 9 data sets.

The inference time was tested for different amounts of workers w. This parameter determines how many component models are inferred in parallel. For most, the run time of $w = 2$ inference was a big improvement over $w = 1$. For $w = 3$ and $w = 4$ the decline in run time was much less steep. The decrease in run time is not linear as some components produced a far greater amount of traces in the logs.

The hybrid determinisation HD_u was run for $u = \{0, 1, 2, ..., 9, 10\}$, The execution time, recall, specificity and balanced accuracy was measured. The determinisation time of $u = 0$ was low for all data sets, except one. $u = 1$ generally had a much larger run time than $u = 0$. Values for all higher u were similar to those of $u = 1$. Recall remained stable for all values of u, and even went up at $u = 1$ slightly for 2 data sets with low recall. Specificity is stable as well, but decreases for 2 data sets. They report that $u = 1$ is the best trade-off between run time and balanced accuracy.

The comparison of PRINS and MINT shows that the PRINS model generally performs similar to MINT in terms of recall. For 2 sets, PRINS' recall was significantly lower. Specificity is significantly higher for PRINS on 3 data sets. The balanced accuracy is significantly higher for one data set and significantly lower for another. As for the model sizes, it was found that PRINS models are on average 3 times as large as MINT models, with 5.5 times as many transitions. For some sets, the amount of states was over 5 times as high, and the amount of transitions over 7 times as high, with one set resulting in models with 19.3 times as many transitions [42].



Methodology

The goal of this thesis is to evaluate and compare software behaviour models created by various techniques in FlexFringe [52], ProM 6 [48], MINT [53] and PRINS [42]. To make a comparison, the four programs with their various algorithms were used to create models on the same dataset. This section will go into what the experimental set-up looks like and why it was done this way.

4.1 Tools and Settings

This evaluation will use several external programs to create and evaluate models. This section will discuss these programs and their settings. Both model inference and model evaluation were run on a `Windows 10` machine with an `Intel Core i7-7700HQ` processor and `16GB` RAM. All model creation jobs and alignment jobs were run in sequence, without anyone using the machine.

4.1.1 Model Inference

FlexFringe

FlexFringe's [52] source code can be downloaded and be made into an executable with `Cmake`. With this executable FlexFringe can be run from the commandline with arguments for the input data, output file and an `.ini` file with the settings, which can be found in appendix A.1. FlexFringe (FF) will be run with the `AIC` heuristic. Symbols and states that are infrequent will not be filtered. Parameter `largestblue=1` speeds up the inference by not considering all blue states, just the most frequent ones.

ProM 6.11

The ProM [49] tool[1] is free and version 6.11 was released in 2021. ProM can be used via a GUI, but this is not suitable for batch processing. Therefore, the ProM plugins are called from the ProM Command Line Interface (CLI) with a Java-like script. The command line java call and an example script with one miner can be found in appendix A.2. The miners are called with their standard parameters, the exception being the inductive miner with the 'infrequent all operators' (IMfa) settings. The values of the standard settings for each miner can be found in table 4.1. Note that noise thresholds in the inductive miner and directly follows miner are not defined the same way.

PRINS and MINT

PRINS is a program written in Python, its source code is in a public project on Github[2]. It uses MINT internally, which comes packaged with its source code[3] in a `.jar`. The PRINS code parses input to a format suitable for MINT and runs the java program by calling it from a python script. The PRINS python code can thus be used to run both PRINS and MINT. A python script is used to run all desired jobs for PRINS and MINT.

[1]http://promtools.org/doku.php?id=prom611
[2]https://github.com/SNTSVV/PRINS
[3]https://github.com/neilwalkinshaw/mintframework

MINT is run with the `AdaBoostDiscrete` classifier and minimum merge score $k = 2$. PRINS is run with worker amount $w = 4$ and with the authors' new *hybrid determinisation* with state merge limit parameter $u = 1$. PRINS was also run without determinisation, which returns an NFA.

Program	Configuration	Method	Parameter	Value
FlexFringe	**AIC**	Akaike Information Theory		
ProM	**IM**	Inductive miner (IM)	Noise threshold	0
	IMfa	Inductive miner (IMfa)	Noise threshold	0.10
	Hybrid-ILP	Hybrid Integer Linear Programming miner	Discovery algorithm	Mine a place per causal relation (flexible heuristics miner)
			Constraints	No trivial regions Empty after completion Theory of regions
			Filter	Sequence encoding
			Filter threshold	0.25
	DF	Directly follows miner	Noise threshold	0.80
MINT	**ADB-2**	AdaBoostDiscrete	min merge score: k	2
PRINS	**W4-HD1**	MINT	Worker amount	4
		Hybrid Determinisation	HD u	1
	W4-NFA	MINT	Worker amount	4

Table 4.1: Parameters of the inference methods

4.1.2 Evaluation Tools

The run calls to the model inference tools, data preparation and evaluation were implemented in `Python`, mostly utilizing well-known libraries such as Pandas. To assist in analysis of FSMs, `NetworkX` [45] was used. This is a package that defines graph objects and implements well-known algorithms for graph analysis.

The 'Process Mining for Python' (`PM4Py`) [3] library was used to construct and analyse Petri nets. This library was used to import and export Petri net (`.pnml`) and `.xes` files, run token-based replay and find alignments.

4.2 Data Sets

The data that will be used consists of 9 software execution logs in `csv` format. These are the nine logs used by Shin et al. [42] in the PRINS project. Shin et al. [42] took five of these sets (Hadoop, HDFS, Linux, Spark and Zookeeper) from Loghub [20], parsed them with Drain [19] and MoLFI [35] and manually refined them. Working versions of Drain and MOLFI can be found in the Logpai logparser project [56]. The other four logs (CoreSync, NGLClient, Oobelib and PDApp) were collected by Shin et al. from a personal computer. As these are logs of software executions, they only contain examples of event sequences that can happen, i.e. there are no negative examples available. Each dataset represents a system. The system log contains multiple execution logs, denoted by the `logID`. An execution log represents a trace that contains log entries. Each log entry is a line containing information such as the date, component or event identifier `tid`. An example can be seen in 4.2

logID	lineID	date	time	level	component	message	tid	template	values
1	1	16/04/07	10:46:05	INFO	ApplicationMaster	"Registered signal handlers for [TERM, HUP, INT]"	E22	"Registered signal handlers for [TERM, HUP, INT]"	[]

Table 4.2: Example of a log entry, taken from the Spark system log [20]

4.2.1 Data Preparation

As the data sets are taken from the PRINS project, they can be used as input for PRINS and MINT without much issue. However, PRINS was created on Linux and generates directories during run-time with component names. Windows naming conventions are more restrictive than Linux's, resulting in errors when a forbidden character is used to create a directory. To prevent this, the forbidden characters are replaced with an underscore beforehand.

Flexfringe only requires an `id` and a `symb` column, which correspond to the `logID` and `tid` column respectively. The input for FlexFringe is parsed to only contain these columns.

ProM input is best given as `XES` format. `XES`, or eXtensible Event Stream, is an `XML` like file format describing the process structure [49]. Each `XES` file contains at least a `log` tag that contains one or multiple `trace` tags. Within these `trace` tags are `event` tags containing the information of a log entry. Converting a `csv` to `XES` can be done within ProM, but the function has also been implemented in `PM4Py` [3], making batch processing easier. Before it can be converted, a timestamp column with a date and time needs to be created. The timestamp is converted to a unified format. After this, `PM4Py` can be used to export the data to XES format using the `logID` and `tid` as `case_id` and `activity_key` respectively.

4.2.2 Data Analysis

The input data was analysed to give an idea of its composition and to investigate whether there is a relation its composition and inference performance on it. Table 4.3 gives an idea of how the data looks in terms of number of traces, unique events and trace lengths. Along with these, diversity of the data will be measured with:

- The average Levenshtein [31] similarity $\overline{S}$ between all pairs of traces. This similarity is an edit distance that counts the minimum amount of deletions, insertions or substitutions needed to transform some trace t_0 in to t_1. The similarity is calculated as:

$$1 - \frac{dist}{max\big(len(t_0), len(t_1)\big)},$$

  meaning identical traces get a score of 1. Two traces of different lengths would need many insertions to transform t_0 into t_1, thus this metric also reflects deviations in trace length.

- Normalized entropy $H_\eta(E)$ [41], where E is the set of unique events, is used to measure the diversity of the event probabilities $p(e_n) = \frac{\text{number of } e_n \text{ occurrences}}{N}$. Normalisation is done using the maximum entropy H_{max} [22], which is calculated with the total amount of events N occurring in the dataset:

$$H_\eta(E) = -\frac{1}{\ln N}\sum_{n=1}^{N} p(e_n)\ln p(e_n),$$

  where an outcome of 1 reflects a perfectly uniform distribution of events.

Log	#Traces	#Events	$\lvert\overline{T}\rvert$	$\sigma_{\lvert T\rvert}$	#Log entries
CoreSync	1418	204	21.3	98.4	30223
Hadoop	68	41	52.6	0.7	3575
HDFS	1000	16	18.7	3.8	18741
Linux	42	115	268.1	243.0	11259
NGLCLient	42	70	21.2	23.3	892
Oobelib	250	147	226.2	61.3	56557
PDApp	787	75	60.2	54.2	47394
Spark	217	21	312.1	489.2	67725
Zookeeper	36	40	702.7	1403.1	25298

Table 4.3: Data charcterisicts: number of traces and unique events, average trace length $\lvert\overline{T}\rvert$ and its standard deviation $\sigma_{\lvert T\rvert}$, number of log entries

4.3 Model Analysis

To analyse the inference tools, k-Folds cross-validation will be used to infer models on training sets. After this, the output models are parsed into a uniform format. With the test set and the inferred models, general metrics, such as the F_2-score, trace fitness, and perplexity will be computed. Runtime and model complexity were also measured.

4.3.1 k-Folds Cross-Validation

k-Folds cross validation is a technique widely used in machine learning to avoid overfitting and utilise every part of the data. It is also helpful when there is no reference model, i.e. a ground truth, which is the case in this research. This technique divides the data into k partitions, as seen in figure 4.1, one partition is labeled the test set and the other partitions will form a training set. The model is inferred using the training set and evaluated with the test set. This process is repeated k times, each time with a different partition as test set and the remaining sets as training set [5, p. 32-33].

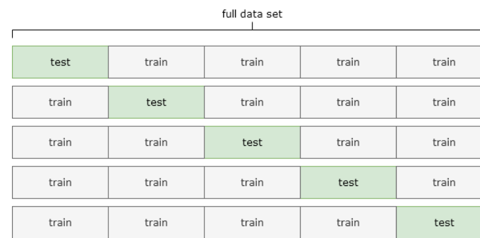


Figure 4.1: k-folds division of a data set with $k = 5$

This method was also used for evaluation by Shin et al. [42] and Walkinshaw et al. [53]. Applying k-folds to this data means considering each trace as a data point, and making k partitions containing $\frac{\#T}{k}$ traces, where $\#T$ is the total number of traces in the data set. Then, each model inference technique will be run k times on each unique training set, resulting in k models for each data set.

4.3.2 Parsing and Conversion

The FSM output is parsed into a NetworkX graph. Models inferred by all programs are parsed to a uniform format. The event labels for transitions are correctly assigned as edge attributes and the accept states are identified and stored in their respective node attributes. During parsing, the transition probabilities are also calculated and stored.

To compute alignments for FSMs, they are converted into Petri nets, as done in the work of Habben-Jansen [18].

4.3.3 Synthesising Negative Logs

Most metrics require *true positive* (TP), *false negative* (FN), *true negative* (TN) and *false positive* (FP) counts from the test set. A TP or TN classification for a trace means it has been correctly classified. A false negative is a trace that should be classified as positive by the model, but is not. A false positive is an invalid trace that should have been classified as negative. The data sets do not contain negative examples and thus cannot retrieve the TN and FP. To solve this, negative logs will be synthesised in the same way as by Shin et al. [42] and Mariani et al. [33].

This is done by randomly selecting positive traces. The amount of traces that will be selected is equal to the amount of traces in the test set. For each trace, a random event will be either *swapped*, *deleted* or *added*. The choice between the actions is random. To make sure ProM reads the trace in the same order, the *swap* operation will also swap the timestamps assigned to the swapped events. After any operation, the trace is sorted by event timestamps. This is done until the trace does not match any positive trace in the dataset. This ensures we end up with a *positive* test set and a *negative* test set of the same size.

4.3.4 General Metrics

To evaluate the models, the TP, FN, TN and FP will be counted through token-based replay for the Petri nets. A simple breadth-first search to follow a path has been implemented for automata. With

these counts, the recall, precision, accuracy and specificity can be calculated. These metrics can be used to determine the balanced accuracy BA and the F_2-score. The different fitness metrics will be calculated with the *positive* test set.

4.3.5 Token-Based Fitness

To identify which traces are correctly classified, they need to be replayed on the inferred model to determine whether the traces end in an accept state. This cannot be done with `PM4Py` as the the converted Petri nets fall victim to the limits of its implementation due to their characteristics. The algorithm struggled to find the proper path through the model. The exact culprit could not be found, but it appears to be caused the combination of multiple transitions with the same label and the silent τ transitions inserted at the start and end.

However, it is possible to simulate this algorithm for automata through a Breadth-first search (BFS). Following a given trace with BFS, it is possible to determine whether a trace complies with the model and to calculate token-based fitness. This fitness is calculated by keeping trace of *produced* p and *consumed* c tokens, and calculating *missing* m and *remaining* r tokens. In a Petri net, each transition consumes one token in order to be able to fire; A token is produced for each outgoing arc of a transition.

In terms of automata, there are no firing rules and the concept of a token does not exist. If a transition has the right label, it can always fire. A transition in an FSM cannot have multiple outbound arcs, so no more than one token can be produced. Therefore each FSM transition will both consume and produce one token: $p = c$. p effectively denotes how many transitions have been traversed. Missing tokens for a Petri net are inserted if some transition is missing a token it needs to fire. The amount of missing tokens will exceed 1 if a transition has multiple inbound arcs from places with no token. It can also increment when the right transition can be reached from a marking, but its input place has no token. As the FSM transition neither needs a token to fire nor has multiple inbound arcs, a missing token will only be inserted a trace when does not end in an accept state. An FSM never produces more than one token, so m will be 1 if a trace is not valid, 0 otherwise.

All of these characteristic of FSM token-replay simplify the calculation f_{FSM} for some trace t:

$$
\begin{aligned}
& r = p + m - c && \text{with } p = c \\
\Rightarrow\quad & r = p + m - p = m \\
& f_{FSM}(t) = \frac{1}{2}(1 - \frac{m}{c}) + \frac{1}{2}(1 - \frac{r}{p}) \\
\Rightarrow\quad & f_{FSM}(t) = \frac{1}{2}(1 - \frac{m}{p}) + \frac{1}{2}(1 - \frac{m}{p}) \\
\Rightarrow\quad & f_{FSM}(t) = 1 - \frac{m}{p} && \text{where } m = 0 \text{ or } m = 1.
\end{aligned}
$$

The relations that must hold for token-replay still hold here: $c \leq p + m$ and $m \leq c$ [2]. Pseudo code of the token-replay can found in appendix B.2.

4.3.6 Alignment-Based Fitness and Cost

Alignments were run with the `PM4Py` library, using a time-out of 15 minutes. With the alignments on the positive test set, the average trace fitness and alignment *cost* is calculated. `PM4Py` calculates this cost with standard values for certain modes. It assigns 1 for silent transitions, meaning the converted Petri nets will always have a cost of at least 2. For each non-synchronous move it adds $1,000$ to the cost.

4.3.7 Perplexity

Perplexity on both FSMs and Petri nets is calculated by assigning a probability to each unique event $p(e_n)$. This probability is defined as the count of e_n divided by the sum of counts of all out edges of the state. Unfortunately, only FlexFringe returns these true counts. Therefore, the probabilities of a transition in a state will be defined as the number of outgoing transitions with label e_n divided by the total number of outgoing edges of that state. Smoothing is used to assign a probability to an event that cannot happen in a certain state. This *unseen* probability is computed with the *alphabet*

size of the model and is defined as $\frac{1}{|V|}$. The probabilities of outgoing transitions must sum to 1, so each probability is normalised with $\frac{1}{1+\frac{1}{|V|}}$. This gives the computation for the probability of an event e_n in a certain state:

$$p(e_n) = \frac{|e_n|}{N_e} * \frac{1}{1 + \frac{1}{|V|}},$$

where $|e_n|$ is the amount of outgoing transitions with this label in the current state, N_e is the total number of outgoing transitions in the current state and $|V|$ is the amount of unique events in the model. The unseen probability is added when a test trace cannot be fully replayed on the model. The probability of a trace s in test set S then becomes:

$$p(s) = \begin{cases} \sum_{e \in s} \log_2 p(e) & (1) \\ \left(\sum_{e \in s} \log_2 p(e)\right) + \log_2 p(e_{unseen}) & (2), \end{cases}$$

where case (1) is a trace that can be fully replayed and ends in accept state. Case (2) is a trace that cannot be fully replayed, stops at an unseen event or does not end in an accept state. The perplexity of a test set S is computed with the trace probabilities $p(s)$:

$$PP(S) = \frac{\sum_{s \in S} p(s)}{|e_{seen}|},$$

where $|e_{seen}|$ is the total number of events that the model was able to replay from all traces in the test set.

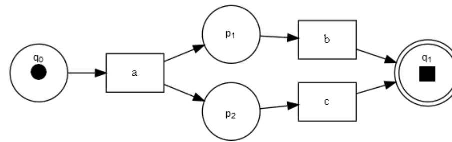


Figure 4.2: A Petri net concurrency relation

The perplexity implementation is trivial for FSMs. After storing the machine into a graph, the probabilities are simply computed by looping though nodes, retrieving their edges, and assigning $p(e_n)$ to each edge. For Petri nets, this requires some additional steps. The concurrency of Petri nets introduces extra paths that are not expressed by the out-degree of places or transitions. A Petri net's reachability graph would express this, but due to the space and time complexity, it is not feasible to convert all Petri nets to reachability graphs. However, it is still possible to compute the perplexity for a Petri net.

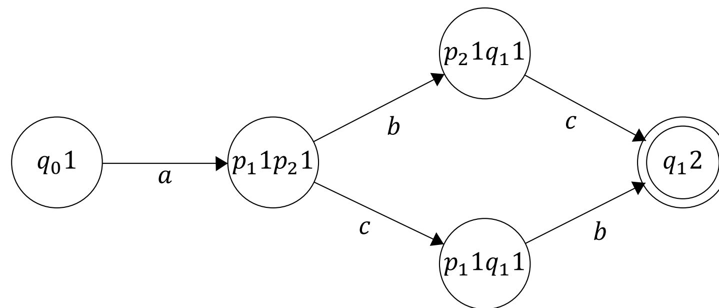


Figure 4.3: Transition graph of figure 4.2

Let us demonstrate this with an example Petri net, seen in figure 4.2. If the implementation for the FSMs would be followed, the probabilities for the transitions would be $p(e_a) = 1$ and $p(e_b) = p(e_c) = \frac{1}{2}$. There are two possible paths in this net: abc and acb. If the probabilities of these paths

is computed with the aforementioned probabilities, both become $p(\langle a,b,c\rangle) = 1 * \frac{1}{2} * \frac{1}{2} = \frac{1}{4}$. But since there are two paths, both should have probability $\frac{1}{2}$. The correct probabilities could be found by transforming the net into a reachability graph. This is done by listing all possible markings and setting them as the states of the reachability graph. A marking M_1 receives an edge with label e_n to M_2 if activating activity e_n when in M_1 leads to M_2 [1, p. 62-64]. The reachability graph of the example Petri net can be seen in figure 4.3, the numbers behind the place labels denote the amount of tokens in the place in this marking. By the definition of $p(e_n)$ the probability of both traces are computed as: $1 * \frac{1}{2} * 1 = \frac{1}{2}$, resulting in the correct probability.

Although computing the reachability graph is not feasible, it is possible to retrieve all travelled markings of a trace. This information is retrieved during the token-based replay. From the definition of the reachability graph follows that: the total number of outgoing edges, is the number of transitions that can be reached from *all* places in the marking. When identifying these edges, all occurrences of the desired transition can be counted, resulting in the correct $p(e_n)$. In the example, the list of travelled markings for trace $\langle a,b,c\rangle$ is: $[[q_0],[p_1,p_2],[p_2,q_1],[q_1]]$. Marking $[q_0]$ only has outgoing transition a, so $p(e_a) = 1$. Marking $[p_1,p_2]$ has two outgoing transitions, e_b, e_c, so $p(e_b) = \frac{1}{2}$ in this marking. After this, one token will be in p_2 and one in q_1, i.e. marking $[p_2,q_1]$. q_1 has no outgoing transitions and p_2 has 1 outgoing transition: c. Therefore, $p(e_c) = 1$ in this marking, resulting in the correct $p(\langle a,b,c\rangle) = 1 * \frac{1}{2} * 1 = \frac{1}{2}$.

This was implemented by modifying `PM4py`'s token-based replay to return a list of all travelled markings. The out edges of each marking were then checked for total occurrences and occurrences of the next transition in the replayed path. The silent transitions were treated the same as the other transitions. This penalises silent transitions in parallel constructs; if a silent transition is present in a parallel construction, it will introduce extra paths that represent the same trace. If a silent transition τ was added to the previous example, it can still only produce abc and acb. The trace $\langle a,b,c\rangle$ can be obtained by travelling through its transition graph in three different ways: $a \to \tau \to b \to c$ and $a \to b \to \tau \to c$ and $a \to b \to c \to \tau$. Each of these paths has a probability of $\frac{1}{6}$ in the reachability graph, which is lower than if the silent transition were not there. Only one will be travelled during token replay, and thus the perplexity for this τ model will be higher. Lastly, it can happen that a marking has multiple outward τ transitions. Since τ is technically *not* a symbol, these will be treated as separate and receive a probability of $p_\tau = \frac{1}{N_e}$.

4.3.8 Cyclomatic and Control Flow Complexity

The cyclomatic complexity (CC) [34] was calculated for the FSMs with:

$$CC = |E| - |V| + 2,$$

Where $|E|$ is the number of edges, $|V|$ the number of nodes. The inference methods do not return several components, so the $2 * |P|$ is omitted. The cyclomatic complexity for Petri nets [44] is computed as follows:

$$CC_{petri} = |F| - (|T| + |P|) + 2,$$

Again, the components are omitted as multiple components in a Petri net means the net is not sound.

The eCFC [6] is computed using the standard calculation for the Petri nets:

$$eCFC_{rel}(Petri) = \frac{eCFC_{abs}(Petri)}{|\{\text{XOR-splits of p}\} \cup \{\text{OR-splits of p}\} \cup \{\text{AND-splits of p}\}|}.$$

For automata, is is calculated with the reduced version:

$$CFC_{rel}(FSM) = \frac{\sum_{q \in Q} |\text{out edges q}|}{|Q|}$$

4.3.9 Run Time

The run times were measured slightly differently for the programs. PRINS has a built-in the measurement in their python script. The run time of the MINT jar is determined from when its Python

subprocess is called to when it returns. This is added to the run times of the rest of their algorithms, that just enclose the function calls.

FlexFringe does not output any time, so its run-time is read in a similar manner to that of the MINT jar. However, since there is only one FlexFringe call from the python script running inference many times in sequence, the script parses the command line output and stores the run time when it encounters certain start and stop lines.

ProM run times are measured as seen in the run script, appendix A.2, using Java's call to the system time.

4.4 Evaluation

This section will give an explanation of the important choices made for running this experiment. The last subsection will elaborate upon choices made in terms of comparing Petri nets to automata.

4.4.1 Data Sets

The decision was made to use the same data sets as Shin et al. [42] for two reasons: the data was available, divers and worked as input for PRINS and MINT. All of these data sets were freely available for research purposes. The 5 Loghub [20] sets are widely used by other research as well.

Secondly, the data has sets with very different amounts of traces, unique events and trace lengths. The 4 sets Shin et al. [42] generated, were meant to diversify sets. They also computed log confidence scores [8] to determine if the sets were suitable for inference.

Lastly, using the same data sets is practical as inference with PRINS requires a specific format for the data, namely a component column and more than 10 traces. Since MINT was included in the PRINS program, these data sets were also be converted internally by the PRINS code to run inference with MINT.

FlexFringe and ProM do not have so many requirements. For FlexFringe, it is enough to simply extract the `logID` and `tid` and rename them to `id` and `symb` respectively. ProM input only requires conversion into `xes` format. This is easily done using `PM4Py` and only needed identification of the date/time format for each set. Some formats required extra parsing, but this can be done with Python's Pandas library.

4.4.2 Data Analysis

The data sets will be analysed in terms of differences one would be able to see and calculate relatively fast before starting inference. These methods should only measure surface level characteristics, as the model inference is the real analysis of the data sets. One would not want to spend much time computing a complex metric on the data set that effectively replaces the need for model inference.

To get an idea of the distribution of the events, the normalized entropy is computed. It is also possible to compute the entropy for traces. However, this would only give an idea of how many of identical traces are in the data set and does not detect minor differences between traces. Therefore, it may be more interesting to investigate the diversity of traces in terms of distance. Since this is the comparison of simple data, namely two sequences of events, the Levenshtein distance was used. This is a simple string comparison metric, that does not require strings to be the same length.

4.4.3 Inference Method and Parameters

Since this work is comparing a total of 8 model inference methods on 9 datasets with 5-folds, the decision was made to not tweak any parameters. The parameters were chosen based on the generally optimal settings reported by their creators.

FlexFringe

FlexFringe includes several PDFA learning algorithms. Verwer and C. Hammerschmidt [51] tested their program on various PAutomac [50] problems. `AIC` infers a PDFA with the lowest perplexity score for majority of the problems, and is close to the best score for the rest.

No filtering will be applied in terms of state and symbol frequency, as this will be applied to a diverse set of data. To reduce run time and increase the likelihood of producing a model for all data sets, `largestblue` is set to 1.

MINT

MINT was evaluated by Walkinshaw et al. [53], using 5 data sets and 5-folds cross-validation. They found that the `AdaBoostDiscrete` algorithm with $k = 2$ produced the best results overall, although performance per data set heavily varied.

PRINS

PRINS' inference and hybrid determinisation technique HD_u was evaluated with the 9 data sets that will be used in this research as well. Setting the worker amount to 4, allows PRINS run MINT inference for 4 components at the same time. This decreases PRINS' run time. Shin et al. [42] found that this did not significantly decrease the accuracy of the models. The u parameter was set to 1, as HD_1 was found to have a good trade-off between run time and accuracy. This tool was chosen as it can use another inference method internally and speed it up. This does come at a cost of potentially generating larger automata [42]. FlexFringe was not inserted into PRINS, as it is so speedy, the state space expansion will not be worth it.

ProM

The ProM miners that were used were chosen from the miners that are in the ProM base package. Three of these were selected by checking the results of the Process Discovery Contest 2021 [38], hosted by the IEEE Task Force for Process Mining. The base inductive miner was added as well. Parameters were left on the ProM standard settings, with the exception of the IMfa miner. The standard noise threshold was lowered, as this research deals with multiple real life logs that are diverse and noisy. The ProM standard settings for the Hybrid ILP miner were in line with the settings that gave good results in the work of Zelst et al. [54].

4.4.4 Negative Log Synthesis

The synthesis of negative logs raises the immediate and valid concern: is it possible that this synthesised trace is positive after all? This method was used by [42] and [33], but no irrefutable evidence was given that these logs are indeed negative. Other works collected their negative traces differently.

Walkinshaw et al. [53] introduced small changes in the code for the program they collected software traces from in their initial experiments. However, they note that this could still result in an unchanged order of events or that the changes were much too obviously wrong. Their second approach was to determine key characteristics for a program and create rules for what cannot happen. With these rules, they synthesised the negative traces.

Mariani et al. [33] tried to validate their negative traces with the Clopper-Pearson and the Agresti-Coull confidence intervals. They synthesised negative traces and replayed them on a model. They kept generating new traces until a 95% confidence level was obtained for the specificity ($\frac{TN}{TN+FP}$).

Traces of correctly programmed software are by definition positive traces. Finding other data sets that are suitable and do contain negative traces is not easy. Creating negative traces manually would require deep knowledge about each of the 9 software systems. It would also take a long time to manually create all rules or traces necessary. The approach taken by [33] uses a rather circular reasoning as it is unknown whether the model is correct and whether the synthesised trace is indeed negative.

Hence, in the scope of this research, it is not possible to obtain negative traces that are certainly negative. To combat some of the irregularities that may arise from using synthesised negative logs, some metrics that rely on the positive test data only were added, such as fitness and perplexity. In addition to this, less importance was assigned to results from the synthesised log by using the F_2-score rather than F_1, thus weighing recall ($\frac{TP}{TP+FN}$) higher than precision ($\frac{TP}{TP+FP}$).

4.4.5 Run Time

The run times are measures slightly differently for each program. In addition to this, Windows tends to background tasks unexpectedly with little control over it. This can influence the run times measured. There is not much that can be done about this; small differences in run times, in terms of tens of seconds will not be considered relevant.

4.4.6 Comparing Petri Nets to FSMs

The first research goal of this work, is determining how to compare Petri nets to FSM output. This section will go into how this was done, why certain choices were made and what the drawbacks are. Some measurements were easy, as they can be done on both Petri nets and FSMs. These will be discussed first. Then, the more complicated measurements, fitness and complexity will be elaborated upon.

Using the metrics such as recall, specificity, precision, accuracy, balanced accuracy and the F-score are common in other studies [42][14][53][13] on automata. The Process Discovery Contest [38] reports on F-scores and accuracy for Petri nets, where the accuracy scores are split into positive and negative accuracy. These metrics rely on counting TPs, FNs, TNs and FPs, which can be done for Petri nets as well as automata; one only needs to check if a test trace complies with the model.

Perplexity was added to the experiment to give another point of view, one computed without the synthesised negative logs. This metric only needs the occurrences of events in a state. This is done for FSMs by counting the occurrence of each label on a transition in a state and can be done in a Petri net by counting this for each travelled marking.

Fitness

Trace fitness was measure to give an idea of how close a machine is to recognizing traces. It is done without the use of the synthesised negative logs, which is another advantage. The fitness is calculated with token-replay and with alignments.

Token-based fitness (TBR fitness) is computed for Petri nets and can be done for automata. However, it has three drawbacks for FSMs especially. First of all, for FSMs the TBR for an invalid trace $m = 1$ becomes: $f_{FSM}(t_{invalid}) = 1 - \frac{1}{p}$, where p is length of the successfully replayed transitions. In figure 4.4 one sees immediately that after 40 compliant events, all fitness scores approach 1. This does not necessarily reflect reality. If a trace had 200 events, with the first 100 compliant, a fitness score of 0.99 will be achieved. A trace with 4 events, with 3 compliant receives a score of 0.75. Thus, this may not be fully representative how well a trace fits to a model.

Secondly, the method does not reflect how 'far' a trace ended from a sink state. When the trace ends, only one more missing token is added to m. If an FSM model and two traces t_1, t_2 are considered, with equal lengths. t_1 ends one transition away from an accept state and t_2 would need to traverse 10 more transitions. These two traces will receive the *same* fitness score. This is a problem for both Petri nets and FSMs and it can be solved by using alignment-based fitness instead.

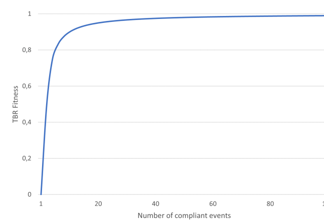


Figure 4.4: Relation between number of events compliant to the model and TBR fitness for $m = 1$ and $p = c$

The last drawback relates to the comparison of Petri nets and FSMs. If a Petri net is missing a token, but does encounter the transition, it keeps playing the rest of the trace. In Petri net terminology, this is an invalid trace on the model. Continuing the replay increases its p and c. Meaning, the overall fitness increases. An FSM stops immediately at the point where a trace is invalid. This could theoretically result in a lower fitness value if the same portion of the trace was valid. However, as seen in our first drawback, there is a chance the FSM fitness comes back high anyway. Luckily, `PM4Py`'s TBR algorithm has a parameter that returns immediately when a non-conforming transition is detected. This does not necessarily mean that the first issue that was discussed for FSMs will also arise for Petri nets. The TBR fitness can still be reduced if there are many remaining tokens, i.e. $p > c$. Also, a transition with multiple incoming arcs can have more than one missing token. If this is not the case, non-representative high fitness scores can be achieved in the same way as for the FSMs. Moreover, the TBR of a petri net will also produce tokens when traversing silent transitions, meaning its p is often larger than the length of a replayed trace.

One may wonder, why calculate the token-based fitness at all? First of all, TBR fitness does give some clue about how many events can fit to a trace. Several sets have relatively short trace length, thus allowing for larger differences. Furthermore, all f_{FSM} have the same flaw and can be compared, although the differences for the sets with longer traces will be small. To add further information about the replay, the proportion of path that fits in terms of the total path size was added and will be reported alongside the TBR fitness score: $\frac{\text{\#model compliant events}}{\text{\#test path size}}$. Note that this proportion will be 1 if the path was fully replayed but did not end in a sink, had missing tokens, or had remaining tokens.

However, the main reason to give TBR fitness a try, is that computing alignments is time and space consuming. This experiment needs to align 3,860 traces, and some of them are very long. If many of the traces are (almost) compliant and the model does not contain much parallelism, this would not be a problem. However, many alignments could not be computed within a reasonable time or space. With the timeout that was set, alignment should take about a day for each data set at maximum. In reality, `PMyP4`'s alignment calculation can only be stopped once per loop and this can take hours. Time-outs and out-of-memory errors mean it may not be possible to obtain all alignment data. Because of these two reasons, the TBR fitness was calculated for analysis as well.

For the alignment based fitness, automata were converted to Petri nets. There should be no reason for this metric to have a different meaning for the automata and Petri nets. The jury is still out on whether reporting the TBR fitness will provide any useful insights. Due to the time efficiency of this metric, it will be computed anyway, and its usefulness will be discussed in chapter 6.

Complexity

To determine readability and complexity, the CC and $eCFC$ were chosen. Two metrics will somewhat supplement each other's shortcomings. The CC is a good metric for automaton complexity, but it does not capture complexity for Petri nets. This is mostly caused by a the concurrency structures a Petri net has. However, the calculating the CC for a Petri net is still a good indicator for visual clutter, as a net with many more arcs than transitions and places will look like a spaghetti model. The reachability of the Petri nets will not be calculated, as this is not feasible for the amount and size of Petri nets that will be generated due to the run time of constructing such a graph [24].

Since it is still desirable to express the complexity of the Petri nets somehow, Cardoso's control flow complexity metric was computed as well. The idea of this metric is to express how many paths some place p spawns on average. For the automata, this metric may not reflect its complexity well, as complexity rises most with the OR-splits, which are not present. Since such a great importance is given to the concept of splits in this metric, it is possible for an automaton or a Petri net with high CC to still get a low $eCFC$ score.

Furthermore, neither of the two metrics expresses impact of the parallel execution on readability. It is more complicated to follow a trace when concurrency occurs, as one needs to remember the extra tokens in other places. On the other hand, this parallel structure does decrease the visual clutter significantly, which impacts readability positively.

Therefore, both complexity metrics will be reported, although one must be mindful of their limitations. The CC will not be fully informative for Petri net complexity, and the $eCFC$ may not reflect complexity of automata at all.

5 Results

This chapter presents all results obtained by the experimental runs. Firstly, the run time of the model inference will be shown in a set of graphs. Then, the size and complexity of the created models will be reported. After this, correctness of the models in terms of soundness, F_2-score, balanced accuracy and other general metrics. This subsection also contains the perplexity scores and the token- and alignment-based fitness. Lastly, the data analysis results will be reported in relation to F_2-score and perplexity.

327 models were created in total and 156 state machines were converted into Petri nets. Some configurations did not create all models within 4 hours or crashed with memory errors. All configurations that did not generate models for the Oobelib set failed because of a timeout. Incomplete runs for `W4-HD1` and `Hybrid-ILP` were also due to timeouts. The incomplete runs on Spark were due to out-of-memory errors. Table 5.1 gives an overview of which configurations completed successfully for each each data set. A full report of all results can be found in appendix C.

Log name	FF	PRINS		MINT	ProM			
	AIC	W4-HD1	W4-NFA	ADB-2	DF	IM	IMfa	Hybrid-ILP
CoreSync	100%	100%	100%	100%	100%	100%	100%	100%
Hadoop	100%	100%	100%	100%	100%	100%	100%	100%
HDFS	100%	100%	100%	100%	100%	100%	100%	100%
Linux	100%	100%	100%	100%	100%	100%	100%	100%
NGLClient	100%	100%	100%	100%	100%	100%	100%	100%
Oobelib	100%	0%	0%	0%	100%	100%	100%	0%
PDApp	100%	80%	100%	100%	100%	100%	100%	100%
Spark	100%	60%	40%	60%	100%	100%	100%	100%
Zookeeper	100%	100%	100%	100%	100%	100%	100%	20%

Table 5.1: Overview of 5-folds experiment: model inference success

5.1 Run Time

Figure 5.1 shows how long model inference took for each configuration on each data set. Any run time under 10^2 seconds can be considered very low. FlexFringe `AIC` completed inference very fast on all data sets, and so did all ProM configurations, with the exception of Hybrid-ILP. `DF` reports such small run times for Hadoop and NGLClient, it mostly fell of the graph. The differences between these five configurations were not significant. One PRINS results immediately attracts attention: `W4-NFA` appears to be slower than `W4-HD1` on the PDApp, Hadoop and NGLClient data set. However, as reported in table 5.1, `W4-HD1` did not run succesfully for PDApp on 2 out of the 5 folds due to a timeout. The Hadoop and NGLClient run times are so small, that the difference is not significant. `ADB-2` is always slower than any other configuration for the rest of the sets. The differences are large for PDApp, Spark and Zookeeper.

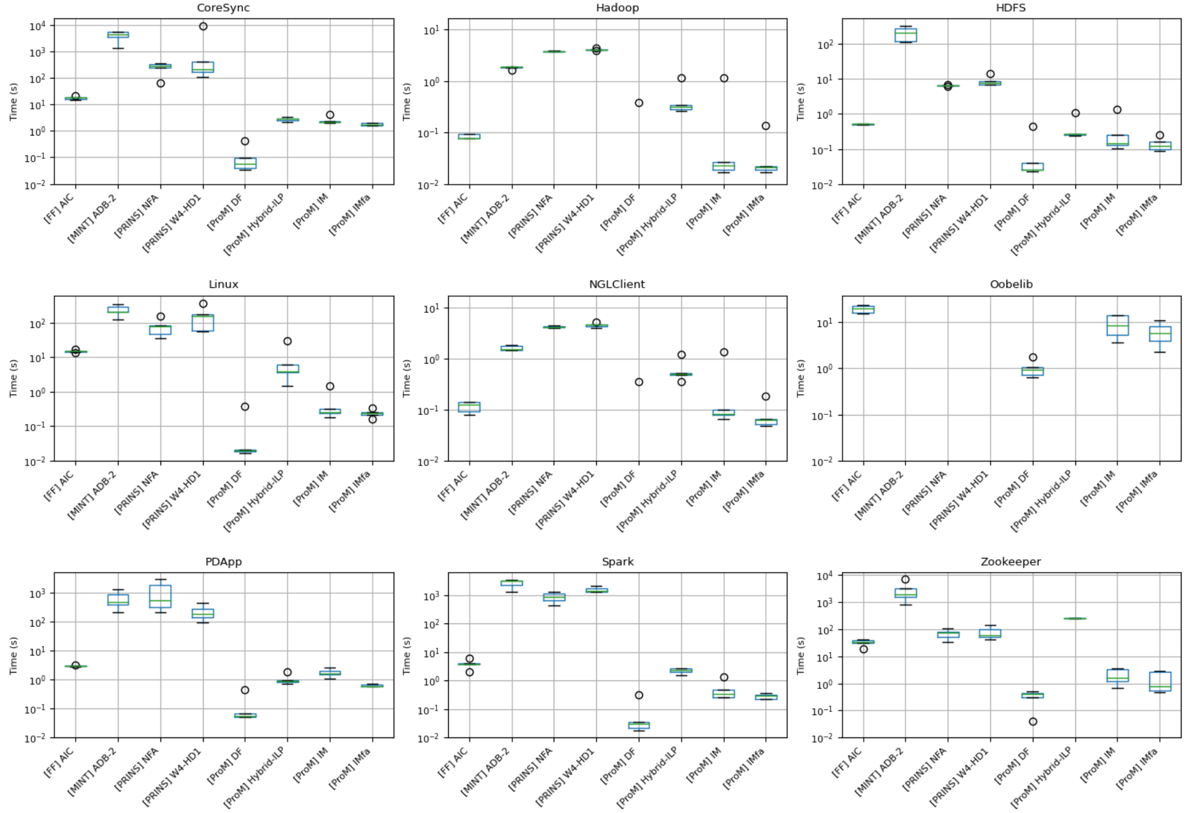


Figure 5.1: Run times of all data sets

5.2 Complexity of Models

The complexity of the models was measured with the CC and the eCFC. This section shows the complexity, as well as the model sizes in terms of states and transitions, in table 5.2. The lower values are bolded, this is done separately for Petri nets and automata.

FlexFringe `AIC` has the lowest amount of states of any automaton in 7 of the 9 data sets; and it is not far behind in the Spark data set and wins by default for Oobelib. It also performs consistently better in terms of the CC. The $eCFC$ for automata is not terribly informative; the results are close to each other, however PRINS `W4-HD1` reports a relatively high $eCFC$ on HDFS.

ProM `Hybrid-ILP` consistently infers small models compared to other configurations. In fact, CoreSync's model is extremely small when one considers that the average number of unique events in its test sets is 153. ProM `DF` reports similar values for HDFS, Hadoop, Spark and Zookeeper. For other data sets, `DF` is firmly in second place. `IMfa` generally produces smaller models than `IM`.

`Hybrid-ILP` and `DF` generally produce less visually cluttered results (lower CC), although `IM` and `IMfa` perform better on Zookeeper and Linux, and achieve similar results on HDFS, Hadoop, Spark. The CC scores of `IMfa` are a fair bit below those of `IM`, except for Zookeeper, where they are practically equal.

The $eCFC$ is generally low for all ProM configurations except `Hybrid-ILP`; it has extremely large spikes compared to the other ProM methods. `IM` has the lowest $eCFC$, `IMfa` either has a slighty higher or equal $eCFC$. `DF` also reports low $eCFC$ scores; they are slightly higher than `IM`'s for Zookeeper, Linux, HDFS and Oobelib.

Log name	Program	Run config	States	Transitions	$eCFC$	CC
CoreSync	FF	AIC	**500.8**	**822.4**	1.6	**323.6**
	MINT	ADB-2	3957.6	4794.0	**1.2**	838.4
	PRINS	W4-HD1	7194.0	10853.4	1.5	3661.4
		W4-NFA	17341.8	20139.2	**1.2**	2799.4
	ProM	DF	**51.6**	**70.6**	**1.2**	21.0
		Hybrid-ILP	**8.6**	**8.2**	**1.2**	**4.0**
		IM	547.6	700.2	**1.2**	402.2
		IMfa	419.2	527.8	1.4	347.0
HDFS	FF	AIC	**86.4**	**164.2**	**1.9**	**79.8**
	MINT	ADB-2	**55.8**	**186.2**	3.3	132.4
	PRINS	W4-HD1	537.6	4130.8	6.1	3595.2
		W4-NFA	5839.8	9381.8	**1.7**	3544.0
	ProM	DF	**10.2**	24.4	1.4	16.2
		Hybrid-ILP	**8.0**	**9.0**	2.8	**9.0**
		IM	51.0	64.2	**1.2**	34.4
		IMfa	22.0	29.2	1.3	18.4
Hadoop	FF	AIC	**53.0**	**54.0**	1.0	**3.0**
	MINT	ADB-2	139.0	144.8	1.0	7.8
	PRINS	W4-HD1	**65.0**	**68.0**	1.0	**5.0**
		W4-NFA	2149.0	2315.2	1.1	168.2
	ProM	DF	**43.0**	49.2	1.1	**8.2**
		Hybrid-ILP	**41.0**	**42.0**	1.3	17.0
		IM	**43.0**	47.0	1.1	**8.0**
		IMfa	**43.0**	47.0	1.1	**8.0**
Linux	FF	AIC	348.0	696.4	2.0	350.4
	MINT	ADB-2	**272.8**	**419.6**	1.5	**148.8**
	PRINS	W4-HD1	1742.8	5147.2	2.7	3406.4
		W4-NFA	1535.4	2401.0	1.6	867.6
	ProM	DF	115.4	298.2	1.4	184.8
		Hybrid-ILP	**37.2**	**22.0**	249.1	417.4
		IM	219.4	215.2	**1.1**	110.2
		IMfa	181.8	163.2	**1.1**	**85.8**
NGLClient	FF	AIC	**91.0**	**106.0**	1.2	**17.0**
	MINT	ADB-2	136.6	146.6	**1.1**	**12.0**
	PRINS	W4-HD1	140.6	175.6	1.2	37.0
		W4-NFA	508.6	650.6	1.3	144.0
	ProM	DF	48.8	64.4	**1.1**	17.6
		Hybrid-ILP	**18.8**	**18.0**	1.2	**12.6**
		IM	135.8	162.2	1.2	100.4
		IMfa	97.6	117.0	1.2	71.0
Oobelib	FF	AIC	**616.2**	**822.8**	1.3	**208.6**
	MINT	ADB-2	-	-	-	-
	PRINS	W4-HD1	-	-	-	-
		W4-NFA	-	-	-	-
	ProM	DF	**91.0**	**199.4**	1.4	**110.4**
		Hybrid-ILP	-	-	-	-
		IM	380.8	449.8	**1.2**	305.8
		IMfa	270.6	320.2	1.3	205.2
PDApp	FF	AIC	**404.8**	**472.2**	1.2	**69.4**
	MINT	ADB-2	1068.6	1241.6	1.2	175.0
	PRINS	W4-HD1	3048.2	3423.0	**1.1**	376.8
		W4-NFA	29374.6	33737.2	**1.1**	4364.6
	ProM	DF	40.0	68.0	1.3	**30.0**
		Hybrid-ILP	**19.0**	**19.0**	12.4	68.6
		IM	228.8	291.4	**1.2**	173.0
		IMfa	114.6	143.2	**1.2**	86.6
Spark	FF	AIC	39.8	53.2	1.3	15.4
	MINT	ADB-2	**21.7**	**30.3**	1.4	10.7
	PRINS	W4-HD1	**20.0**	**25.0**	**1.2**	**7.0**
		W4-NFA	1366.0	1826.5	1.3	462.5
	ProM	DF	**17.0**	20.0	**1.1**	**5.0**
		Hybrid-ILP	**15.0**	**14.8**	2.3	18.6
		IM	24.0	30.0	**1.1**	10.0
		IMfa	24.0	27.0	**1.1**	**7.0**
Zookeeper	FF	AIC	**280.8**	685.2	2.4	406.4
	MINT	ADB-2	344.8	**574.0**	**1.7**	**231.2**
	PRINS	W4-HD1	1785.4	4741.2	2.6	2957.8
		W4-NFA	2087.2	4381.4	2.1	2296.2
	ProM	DF	**42.0**	172.0	1.6	132.0
		Hybrid-ILP	**44.0**	**36.0**	93.9	271.0
		IM	126.0	157.2	**1.2**	**88.0**
		IMfa	108.8	136.8	1.4	**88.8**

Table 5.2: Complexity in terms of number of states, edges, $eCFC$ and $CC1$

5.3 Correctness of Models

This section will report the results related to the correctness of the models. This is done in terms of soundness, F_2-score and balanced accuracy, TBR and alignment-based fitness and perplexity.

5.3.1 Soundness

All Petri net models were checked for soundness, results are shown in 5.3. The inductive miners infer a sound net by design, the rest was checked with PM4Py. However, for some nets the soundness computation did not complete within a day. The calculation for PDApp on W4-NFA gave a memory error.

Log name	**FF**	**PRINS**		**MINT**	**ProM**			
	AIC	**W4-HD1**	**W4-NFA**	**ADB-2**	**DF**	**Hybrid-ILP**	**IM**	**IMfa**
CoreSync	✓	1 ✗ 4 □	□	✗	✓	2 ✓ 3 ✗	✓	✓
Hadoop	✓	✓	✗	✓	✓	✗	✓	✓
HDFS	✓	4 ✓ 1 ✗	✗	✓	✓	✗	✓	✓
Linux	✓	✗	✗	✓	✓	✗	✓	✓
NGLClient	✓	✓	✓	✓	✓	4 ✓ 1 ✗	✓	✓
Oobelib	✓	-	-	-	✓	-	✓	✓
PDApp	✓	✗	■	1 ✓ 4 ✗	✓	✗	✓	✓
Spark	✓	✓	✗	✓	✓	✓	✓	✓
Zookeeper	✓	4 ✗ 1 ✓	✗	1 ✓ 2 ✗ 1 □	✓	✗	✓	✓

Table 5.3: Soundness of generated Petri nets. □= a time out, ■= a memory error.

5.3.2 F_2-score and Balanced Accuracy

In figure 5.2 all BA and F_2-scores can be found. Additionally, the recall, precision, and specificity are reported in table 5.4. The results for the test data will be discussed in this section per dataset.

CoreSync and PDApp. AIC, ADB-2 and both PRINS configurations report high values for both the BA and the F_2-score. Their F_2-score is always higher than the BA for CoreSync, whereas PDApp reports marginal differences for these four configurations. Table 5.4 shows a precision and specificity lower than the recall for these configurations on CoreSync. For PDApp, the recall values are more similar to the specificity and precision. The best performing configuration by ProM is DF, which reports a BA and F_2-score above 0.8, coming fairly close to the former four configurations. Hybrid-ILP and IM have a large gap between their BA and F_2-scores, which is caused by their recall being much lower than their precision and specificity for CoreSync. The gap for IM on PDApp is due to a poor specificity of 0.166. It should also be noted that Hybrid-ILP, IM and IMfa report high variability on their F_2-scores, but not on their BA, indicating very different recall and/or specificity results across the different folds.

Hadoop. All configurations perform well on Hadoop, and the F_2-score is slightly higher than the BA for most configurations. DF and Hybrid-ILP show some deviation in their BA as well as their F_2-score. Hybrid- ILP performs the worst and DF is in between it and all other configurations.

HDFS. AIC outperforms all other configurations in terms of BA and F_2-scores. Fairly large gaps between BA and F_2-score can be seen for the other configurations, indicating some gaps in recall, precision and specificity. Table 5.4 shows that ADB-2, W4-HD1 and W4-NFA actually have better recall than AIC, but much lower precision and specificity. DF, IM and IMfa have high recall values as well, in contrast to Hybrid-ILP, which has a recall of only 0.237. Its BA is not as low because of its excellent specificity value; Hybrid-ILP reports the highest specificity and precision. Once again, IM reports high standard deviation; in spite of its good average F_2-score, the deviation shows some of the models performed quite poorly.

Linux. In stark contrast to the previous data set, AIC has a recall of 0 and thus an F_2-score of 0. Its precision is also 0, meaning there were no true positives. It did manage to correctly reject 70% of the negative traces. Interestingly, W4-HD1 outperforms ADB-2 and non-determinised W4-NFA. This is because W4-HD1 reports much higher precision and specificity than ADB-2. IMfa has a low

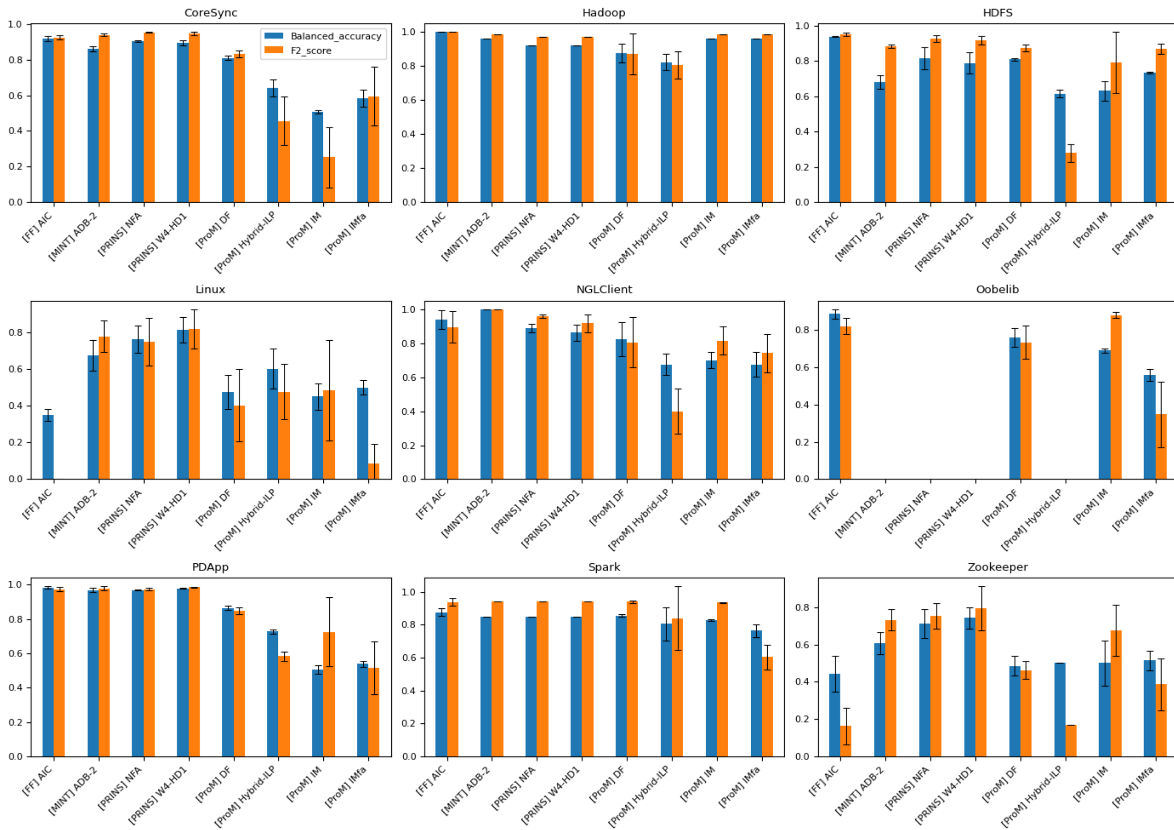


Figure 5.2: Balanced accuracy BA and F_2-score

F_2-score as well, as its recall is close to 0 and its precision is low. Its specificity is very high. IM has a better F_2-score as it does manage to recognise about 50% of the positive traces. Hybrid-ILP has a similar recall to DF and IM, but a higher specificity, resulting in a higher BA. In this data set, all configurations have a significant standard deviation.

NGLClient. ADB-2 performs perfectly for this data set. AIC is not far behind, due to its perfect score on precision and specificity. The PRINS configurations have a higher recall than AIC. W4-NFA reports a perfect recall score, just like ADB-2, indicating some information got lost during determinisation for W4-HD1. Hybrid-ILP lags far behind due to poor recall, but reports a perfect precision and specificity, indicating there were no false positives. DF performs steadily, with results of around 0.8 for all metrics. IM outperforms IMfa across the board. Again, many sets show a lot of variation in BA and F_2-scores.

Oobelib. Only half of the configurations managed to complete inference on this data set. AIC has the highest F_2-score and BA due to very good precision and specificity. However, IM has near perfect recall and outperforms IMfa by a lot in terms of recall and precision, but not in terms of specificity. This indicates that the IMfa did poorly on correctly identifying positive traces.

Spark. An important note for this data set is that, W4-HD1 and ADB-2 inferred only 3 out 5 models and W4-NFA 2. Interestingly, W4-HD1, W4-NFA and ADB-2 have identical performance for all metrics. They also have perfect recall, as does IM. All configurations do well on recall, except IMfa. However, this configuration does have excellent scores on precision and specificity, whereas all other configurations score around 0.7. Hybrid-ILP shows a lot of variation for its BA and F_2-score.

Zookeeper. Again, it should be noted that Hybrid-ILP only managed to create one model successfully. Recall is poor for this model, and precision is only 0.5. It does perform well on specificity, in fact, it is ahead of all other configurations in this regard. AIC performs equally as poorly on recall as Hybrid-ILP, and IMfa did not do well on this metric either, resulting in low F_2-scores for all three. DF also did not perform well, however it has more consistent results across all metrics. IM outperforms IMfa in terms of recall, causing it to have a higher F_2-score. These two configurations have

similar BA scores, as `IM` has a poor specificity score. `W4-HD1` performs the best across the board, with `W4-NFA` close behind. `ADB-2` has a satisfactory F_2-score due to its recall.

Log name	Program	Run config	BA	F_2-score	Recall	Precision	Specificity
CoreSync	FF	AIC	**0.918**	0.925	0.929	**0.909**	**0.907**
	MINT	ADB-2	0.862	**0.939**	**0.986**	0.79	0.738
	PRINS	W4-HD1	0.894	**0.947**	**0.979**	0.836	0.809
		W4-NFA	0.903	**0.952**	**0.983**	0.848	0.823
	ProM	DF	0.81	0.833	0.845	0.79	0.775
		Hybrid-ILP	0.64	0.456	0.421	0.749	0.86
		IM	0.506	0.252	0.236	0.53	0.777
		IMfa	0.583	0.596	0.613	0.571	0.553
HDFS	FF	AIC	**0.939**	**0.951**	0.959	0.922	0.919
	MINT	ADB-2	0.682	0.883	**0.993**	0.613	0.37
	PRINS	W4-HD1	0.79	0.919	**0.994**	0.71	0.585
		W4-NFA	0.817	0.926	**0.99**	0.742	0.643
	ProM	DF	0.81	0.875	0.911	0.758	0.709
		Hybrid-ILP	0.616	0.279	0.237	**0.979**	**0.995**
		IM	0.631	0.791	0.882	0.582	0.381
		IMfa	0.735	0.869	0.941	0.667	0.529
Hadoop	FF	AIC	**1.0**	**1.0**	1.0	**1.0**	**1.0**
	MINT	ADB-2	0.962	0.985	1.0	0.929	0.923
	PRINS	W4-HD1	0.923	0.97	1.0	0.867	0.846
		W4-NFA	0.923	0.97	1.0	0.867	0.846
	ProM	DF	0.877	0.872	0.877	0.88	0.877
		Hybrid-ILP	0.823	0.806	0.8	0.837	0.846
		IM	0.962	0.985	1.0	0.929	0.923
		IMfa	0.962	0.985	1.0	0.929	0.923
Linux	FF	AIC	0.35	0.0	0.0	0.0	0.7
	MINT	ADB-2	0.675	0.778	**0.825**	0.635	0.525
	PRINS	W4-HD1	**0.812**	**0.818**	**0.825**	**0.805**	0.8
		W4-NFA	0.763	0.749	0.75	0.768	0.775
	ProM	DF	0.475	0.402	0.4	0.437	0.55
		Hybrid-ILP	0.6	0.476	0.45	0.629	0.75
		IM	0.45	0.483	0.525	0.379	0.375
		IMfa	0.5	0.084	0.075	0.28	**0.925**
NGLClient	FF	AIC	0.938	0.895	0.875	**1.0**	**1.0**
	MINT	ADB-2	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
	PRINS	W4-HD1	0.862	0.918	0.95	0.809	0.775
		W4-NFA	0.888	0.957	1.0	0.818	0.775
	ProM	DF	0.825	0.804	0.8	0.836	0.85
		Hybrid-ILP	0.675	0.399	0.35	**1.0**	**1.0**
		IM	0.7	0.816	0.875	0.647	0.525
		IMfa	0.675	0.741	0.775	0.647	0.575
Oobelib	FF	AIC	**0.884**	0.819	0.788	**0.975**	**0.98**
	ProM	DF	0.758	0.732	0.724	0.775	0.792
		IM	0.688	**0.877**	**0.98**	0.618	0.396
		IMfa	0.558	0.347	0.324	0.487	0.792
PDApp	FF	AIC	**0.984**	0.974	0.968	**0.999**	**0.999**
	MINT	ADB-2	0.968	**0.98**	**0.987**	0.951	0.949
	PRINS	W4-HD1	0.979	**0.985**	**0.989**	0.97	0.97
		W4-NFA	0.968	0.974	0.978	0.959	0.958
	ProM	DF	0.865	0.848	0.839	0.886	0.892
		Hybrid-ILP	0.727	0.584	0.541	0.858	0.911
		IM	0.505	0.725	0.843	0.494	0.166
		IMfa	0.538	0.516	0.525	0.544	0.552
Spark	FF	AIC	**0.877**	**0.939**	0.977	0.816	0.777
	MINT	ADB-2	0.849	**0.943**	**1.0**	0.768	0.698
	PRINS	W4-HD1	0.849	**0.943**	**1.0**	0.768	0.698
		W4-NFA	0.849	**0.943**	**1.0**	0.768	0.698
		DF	0.856	**0.94**	0.991	0.78	0.721
		Hybrid-ILP	0.805	0.841	0.875	0.757	0.735
		IM	0.83	**0.936**	**1.0**	0.746	0.66
		IMfa	0.765	0.604	0.554	**0.959**	**0.977**
Zookeeper	FF	AIC	0.443	0.162	0.143	0.433	0.743
	MINT	ADB-2	0.607	0.733	0.786	0.581	0.429
	PRINS	W4-HD1	**0.742**	**0.794**	**0.828**	0.738	0.657
		W4-NFA	0.714	0.754	0.771	0.698	0.657
	ProM	DF	0.486	0.462	0.457	0.492	0.514
		Hybrid-ILP	0.5	0.167	0.143	0.5	**0.857**
		IM	0.5	0.674	0.743	0.497	0.257
		IMfa	0.514	0.388	0.372	0.514	0.657

Table 5.4: BA, F_2-score, recall, precision, accuracy and specificity

Log name	FF	PRINS		MINT	ProM			
	AIC	W4-HD1	W4-NFA	ADB-2	DF	IM	IMfa	Hybrid-ILP
CoreSync	99.5%	0%	0%	19.9%	100%	93.5%	78.7%	100%
Hadoop	100%	100%	100%	100%	100%	100%	100%	100%
HDFS	100%	80%	0%	100%	100%	100%	100%	100%
Linux	7.5%	0%	0%	95.0%	97.5%	70%	22.5%	97.5%
NGLClient	100%	100%	100%	100%	100%	92.5%	95%	100%
Oobelib	20%	-	-	-	100%	0%	0%	-%
PDApp	99.9%	0%	0%	79.9%	100%	0%	0%	80%
Spark	78.1%	100%	0%	100%	100%	100%	100%	100%
Zookeeper	11.4%	0%	0%	34.3%	17.1%	25.7%	14.3%	0%

Table 5.5: Percentage of traces that were aligned without out-of-memory or time-out errors

5.3.3 Fitness

In figure 5.3 all fitness metrics can be found. The 'percentage of edges travelled' expresses which portion of a trace was successfully replayed before encountering an impossible event and getting stuck in a state. If no green bar is present, the alignment could not be completed in time or ran out of memory. The percentage of successful alignments can be found in table 5.5. The alignment cost is also reported in figure 5.4.

The first thing of note is that, as predicted, all state machine inference methods report a TBR fitness of 1 or close to it. `AIC` on NGLClient and Linux are the only exceptions to this trend. However, the TBR fitness (f_{TBR}) for these methods does not differ much from the alignment fitness (f_{AL}). Linux, NGLClient, Oobelib and Zookeeper, show a greater deviation in percentage of travelled edges. The

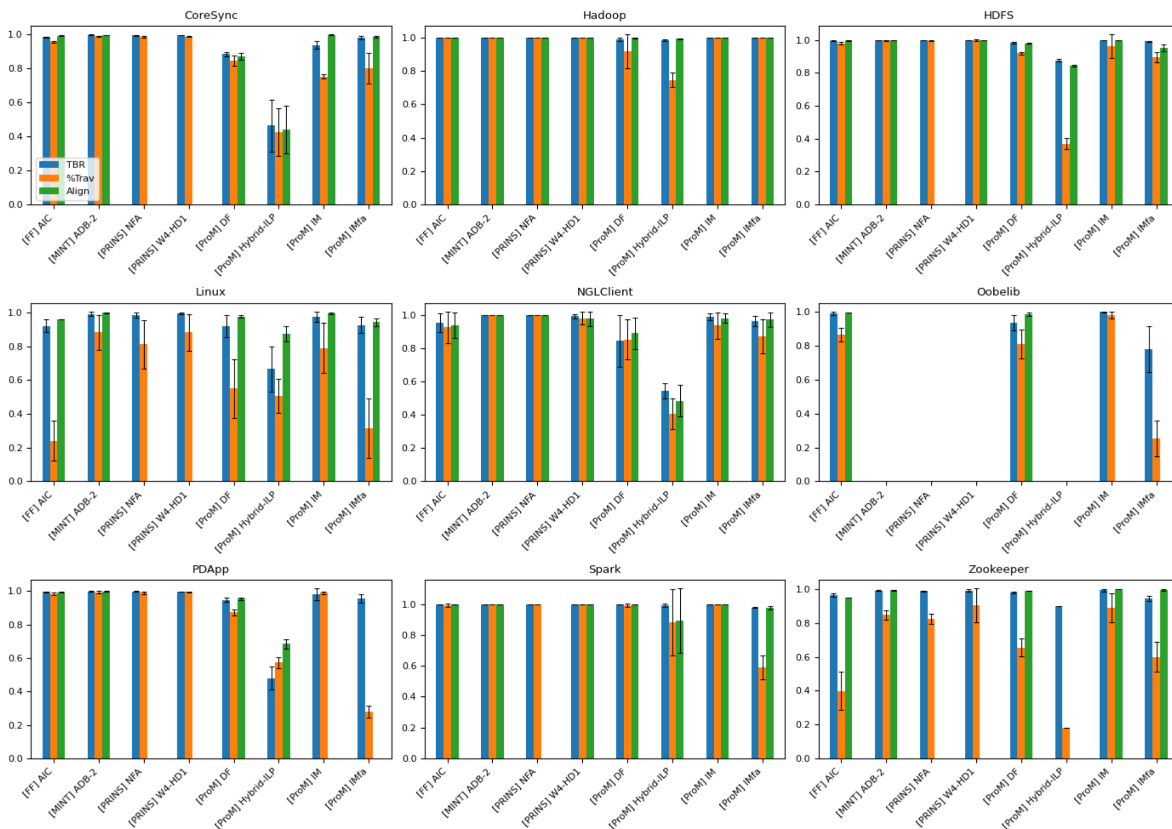


Figure 5.3: TBR fitness f_{TBR}, percentage of edges traversed, Alignment-based fitness f_{AL}

fitness and alignment costs will be discussed per configuration.

AIC. The f_{AL} of the `AIC` configuration does not drop below 0.95 for any of the data sets. The f_{AL} is $0.95 < f_{AL} < 0.99$ for Linux, NGLClient and Zookeeper and $f_{AL} > 0.99$ for the rest. It should be noted that few alignments were computed on Linux, Zookeeper and Oobelib. Alignments for Oobelib completed fully on 1 of the folds, whereas Linux and Zookeeper have a partial result for one of the folds. The percentage of edges traces can successfully traversed on the `AIC` models is also very high, except for Linux and Zookeeper. The cost of the alignments of `AIC` for CoreSync, Hadoop, HDFS, Oobelib, PDApp and Spark are low and only Spark has major deviations. $cost_{AL}$ values of `AIC` on NGLClient peak for two of the folds, and relatively low costs for the rest. Zookeeper and Linux reported only 1 $cost_{AL}$ for `AIC`, and they were fairly high.

ADB-2. Alignments for `ADB-2` were completed to a satisfactory degree for most data sets, except for CoreSync and Zookeeper. CoreSync alignments almost completed for one fold, whereas a completion of approximately 85% was achieved for two of the Zookeeper folds. The f_{AL} for `ADB-2` never drops below 0.99 and the average traversed edges does not drop below 0.85. The cost of alignment is low for all data sets, except Zookeeper.

PRINS. Not many alignments were computed for these configurations, mainly due to memory errors. For `W4-NFA` the only alignments that could be computed, were for Hadoop and NGLClient. `W4-HD1` completed alignments for these data sets and for Spark and 80% of HDFS alignments. `W4-NFA` and `W4-HD1` have similar fitness scores, but the percentage of travelled edges is higher for `W4-HD1` on Linux and Zookeeper. The percentage of travelled edges is always higher than 0.8. Of the alignments that were computed, the f_{AL} was near 1 or 1, as these configurations as achieved (near) perfect recall on some of the data sets. The alignment costs are also low for all computed alignments.

DF. The f_{TBR} for `DF` is high, with small dips for CoreSync and NGLClient. However, these lower fitness values are still around 0.85. Average traversed edges is generally high, except for Linux and Zookeeper. Almost all alignments for `DF` were computed, except for the Zookeeper alignments. f_{AL} is around 0.98 for most sets and around 0.88 for CoreSync and NGLClient. The $cost_{AL}$ is generally high,

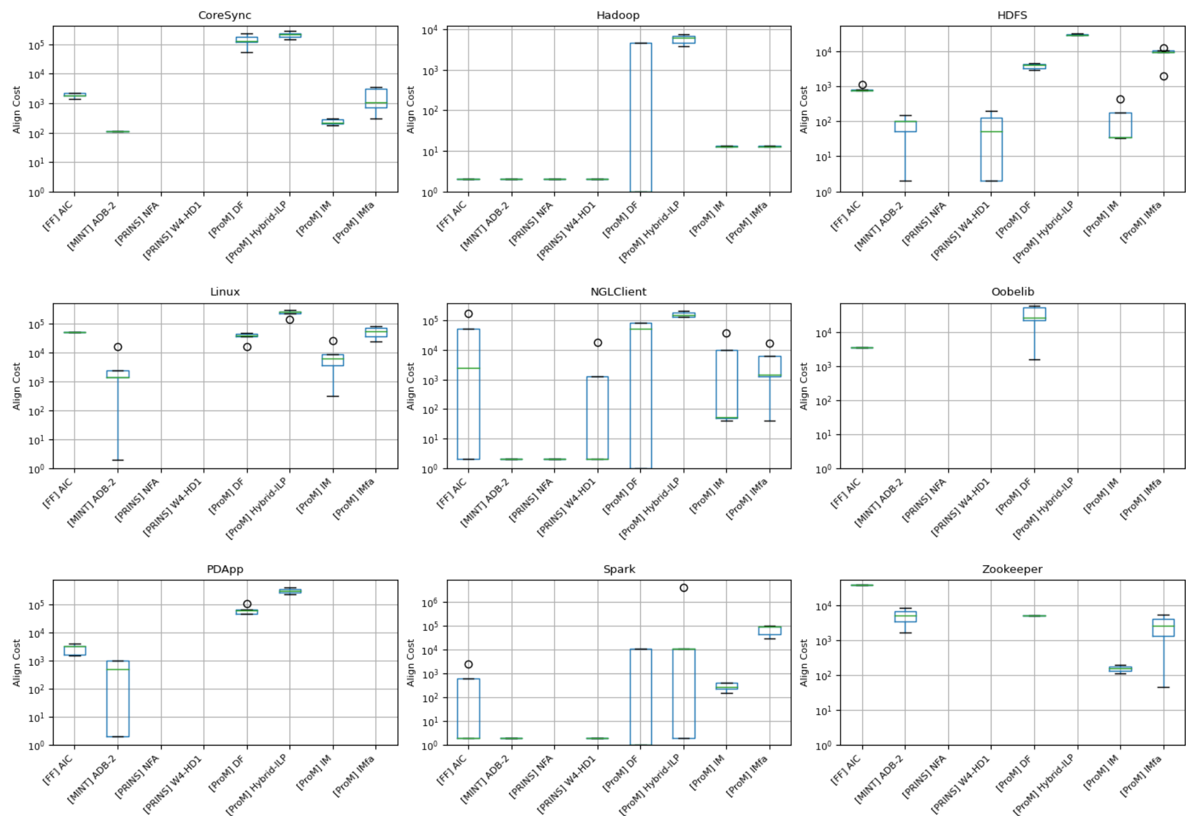


Figure 5.4: Alignment cost $cost_{AL}$

but low for HDFS and Zookeeper. NGLClient and Spark do have several folds where the alignment cost is low.

Hybrid-ILP. This configuration reports a high f_{TBR} for some sets, but for CoreSync, Linux, NGLClient and PDApp the f_{TBR} hovers around 0.5. The percentage of travelled transitions is generally mediocre to low, except for Hadoop and Spark. Almost all alignments for `Hybrid-ILP` were computed successfully, except for Zookeeper. The f_{AL} are lowest for CoreSync, NGLClient and PDApp. The values are generally close to the f_{TBR}, but their are major differences for Linux, PDApp and Spark. The f_{AL} is lower for Spark and higher for the other two data sets. The $cost_{AL}$ is high for all data sets on almost all folds; only Spark has 2 folds that report a low $cost_{AL}$.

IM & IMfa. The f_{TBR} is very high and very similar for these configurations on all data sets, with the exception of Oobelib, where `IMfa` has a lower f_{TBR} than `IM`. The percentage of travelled transitions is generally lower for `IMfa`: its values a poor for for Linux, Oobelib and PDApp. For CoreSync, the travelled percentage is a bit higher for `IMfa` than `IM`. The computation of alignments was not successful for PDApp and Oobelib and few alignments were computed for Zookeeper. Additionally, the Linux alignment for `IMfa` was only done partially for all folds. f_{AL} is very high for both configurations, and always highest for `IM`. $cost_{AL}$ is lower for `IMfa` on NGLClient, equal for Hadoop and much higher for all other data sets. The cost for `IM` is low for all sets an bit higher for Linux. `IMfa` reports very high $cost_{AL}$ for Linux and Spark.

5.3.4 Perplexity

This section reports on the perplexity PP, the results are in table 5.6. A lower score is better, 1 is the lowest score that can be achieved. The amount of parallelism in Petri nets is also reported in figure 5.5, as supplementary information. The `DF` configuration does not contain parallelism by design.

Log name	FF	PRINS		MINT	ProM			
	AIC	W4-HD1	W4-NFA	ADB-2	DF	IM	IMfa	Hybrid-ILP
CoreSync	2.00	2.48	1.93	2.64	**1.82**	109.81	70.35	2.53
Hadoop	**1.06**	1.37	1.33	**1.11**	1.44	1.40	1.40	1.70
HDFS	**1.94**	6.44	4.49	8.03	2.88	10.48	3.01	3.76
Linux	**2.84**	3.79	3.44	4.33	7.55	31.45	19.21	23.79
NGLClient	**1.32**	1.68	1.59	1.49	1.76	38.75	23.61	1.92
Oobelib	**1.46**	-	-	-	4.90	169.54	98.99	-
PDApp	**1.19**	1.54	**1.26**	2.22	2.88	80.99	41.64	7.20
Spark	1.69	2.97	1.99	**1.19**	3.02	2.30	2.61	7.46
Zookeeper	**3.30**	4.73	4.32	5.79	6.02	49.14	34.58	3.97

Table 5.6: Perplexity, lower scores are better.

AIC. In terms of perplexity, `AIC` either outperforms all other configurations, or reports a low perplexity that is close to the best perplexity. The `AIC` perplexity spikes for Linux and Zookeeper, the same data sets where it reported a low percentage of travelled edges $\%Trav$. On the Oobelib set, no other configuration is event close to `AIC`. For HDFS and Linux there is also a large performance gap with most other configurations.

PRINS. The PRINS configurations generally boast a similar performance, but `W4-NFA` is always a bit better. On HDFS and Spark, the gap is larger than for the other sets. The perplexity scores for HDFS, Linux, Spark and Zookeeper are significantly worse for the PRINS configurations than those of the `AIC` configuration.

ADB-2. This configuration performs better than the PRINS configurations on Hadoop, NGLClient and Spark and comes close to or outperforms `AIC` on these data sets. For the other sets, its perplexity is ~2-4 times higher than that of `AIC`.

DF. The perplexity scores for this configuration are generally fairly close to those of `AIC`, but it has a few outliers. It performs much worse on HDFS, Linux and Zookeeper. The perplexities for these sets are also higher than those of the PRINS and MINT configurations. It outperforms all other ProM tools.

IM & IMfa. On CoreSync, Linux, NGLClient, Oobelib, PDApp and Zookeeper, these configurations report very high perplexity scores. These are also the configurations that have higher amounts

of transitions causing concurrency and higher average out degrees, seen in figure 5.5. For the other data sets, `IMfa` outperforms or comes close to the `W4-NFA` perplexity scores. `IM` outperforms `IMfa` on the spark set, and both configurations have the same perplexity for Hadoop. `IM` generally performs the worst. `IMfa`'s scores overall are much better, but still far off the perplexity scores of other configurations.

Hybrid-ILP. This configuration has large spikes in the number of parallel edges for Linux, PDApp, Spark and Zookeeper. The last three of these have a much lower average out degree for the parallel transitions. The perplexity score for Linux is much higher than the rest of the scores. `Hybrid-ILP` performs much worse than the FSM configurations on Linux, PDApp and Spark. For all other sets, its performance is close to that of the PRINS configurations.

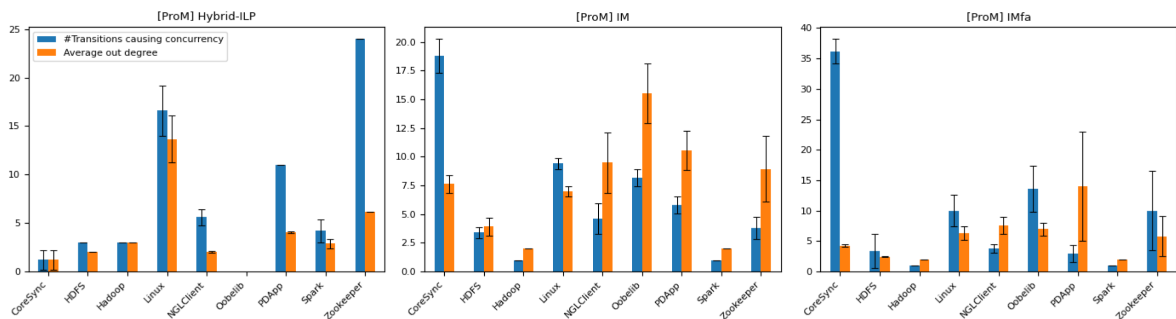


Figure 5.5: Amount of transitions causing concurrency and their average out degree

5.4 Data Composition and Performance

Each test data set was analysed to compute its trace similarity $\overline{S}$ and the normalised entropy $H_\eta(E)$. The results of the trace similarity and event entropy for the full data sets can be found in 5.7. The BA, F_2-score, fitness, perplexity and cyclomatic complexity CC were plotted against trace similarity, normalised entropy and number of unique events. The most interesting graphs are shown in this section, the full results of the analysis can be found in appendix D.

Log	$\overline{S}$	$H_\eta(E)$
CoreSync	0.19	0.410
Hadoop	0.98	0.424
HDFS	0.66	0.209
Linux	0.25	0.232
NGLClient	0.31	0.520
Oobelib	0.63	0.291
PDApp	0.43	0.285
Spark	0.42	0.030
Zookeeper	0.22	0.214

Table 5.7: Data analysis on full data sets

The graph for BA, F_2-score versus the trace similarity $\overline{S}$, figure 5.6, does not show a specific shape to either of the lines. However, for `AIC`, `ADB-2`, `DF`, and the PRINS configurations, it does appear that BA and F_2 performance becomes more stable as the trace similarity grows. The perplexity PP versus normalised entropy $H_\eta(e)$ also does not have a shape that could possibly correspond to a function. However, the perplexity does spike for all sets around a $H_\eta(e) = 0.25$.

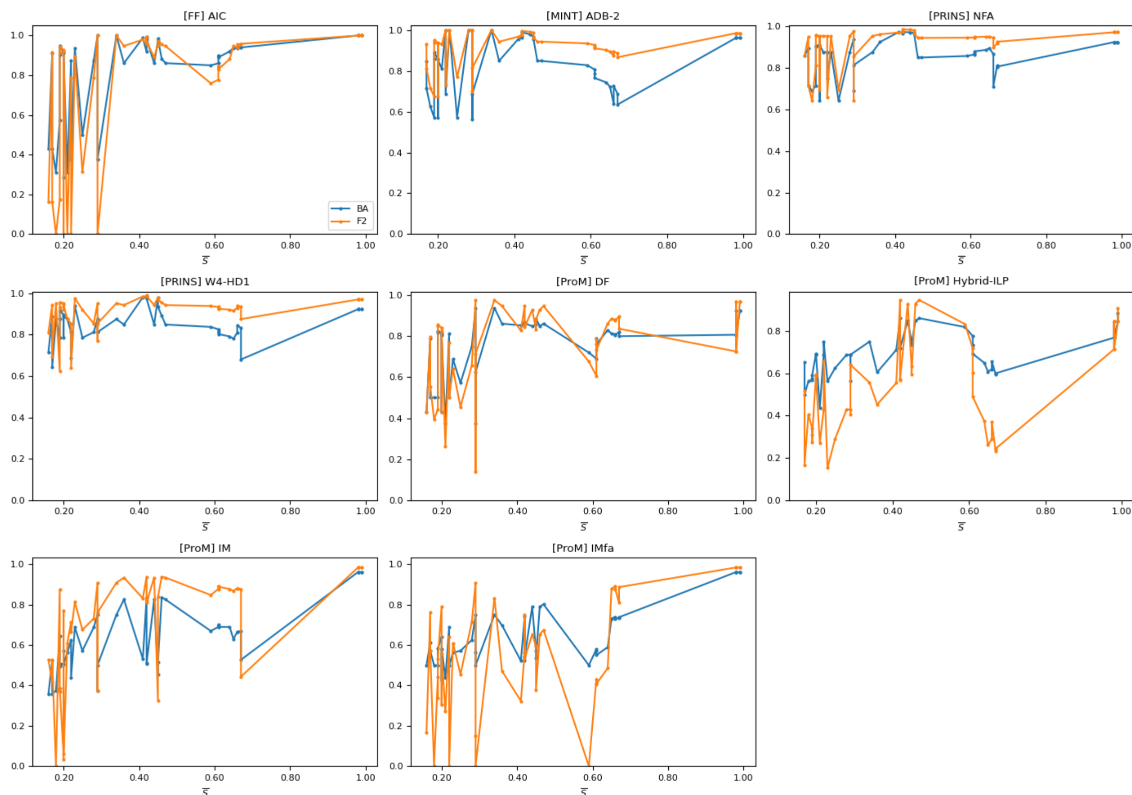


Figure 5.6: BA, F_2-score versus trace similarity $\overline{S}$

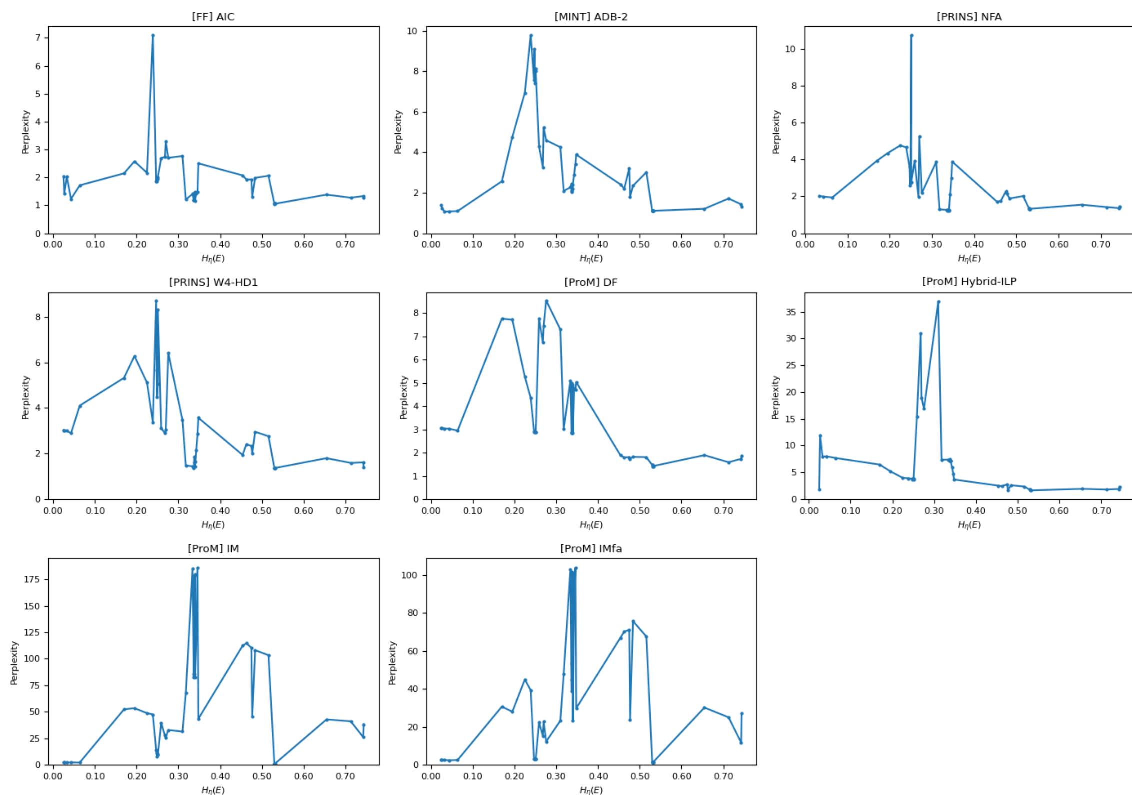


Figure 5.7: Perplexity PP versus normalised entropy $H_\eta(e)$

6
Discussion

This section will discuss the findings of the experiment conducted in this thesis. Firstly, the performance of the metrics on both FSMs and Petri nets will be discussed. Then, the overall correctness of the models and the impact of the synthesised negative traces on these metrics will be considered. After this, the model complexity and inference time will be examined in context of the performance. The analysis of data set characteristics as a possible predictor for performance will then be elaborated upon. Lastly, the limitations of this work will be reported.

6.1 Comparing Apples to Oranges

The first challenge to comparing the various mining and inference tools is to compare the Petri net output to the finite state machine output. Doing a surface-level comparison proved to be straightforward in some regards; the run-time and the F_2-score, BA, recall, precision and specificity can be computed in exactly the same way. However, problems start to arise when attempting to compare the complexity of the models. And even more problems surface when looking deeper into how well traces fit the model; a very important part of model analysis as a model that misses one transition can score low recall, but still fit most of the trace.

These problems stem from characteristics in Petri nets: the time and space complexity of Petri net problems. Namely, the parallel structures in Petri nets. The considered solutions all ran into problems:

1. *Transform Petri nets into FSMs*. This would mean constructing its reachability graph, which can be infinite. Another option would be the coverability graph, a similar construct that can handle loops, but this may not express everything the Petri net could. Furthermore, the 'reachability problem' is thought to be NP-complete [15], and most of the models could indeed not be converted within reasonable time with `PM4Py`.

2. *Transform FSMs into Petri nets*. This still gave issues with the TBR fitness, this was partially due to the TBR implementation of `PM4Py` not being able to handle the converted Petri net. Furthermore, there were time issues while computing alignments as finding an optimal alignment can take a long time. It also did not particularly help the perplexity score problem, as this metric does not express quite the same thing when parallelism is present in the model. This transformation also does not help the model complexity analysis.

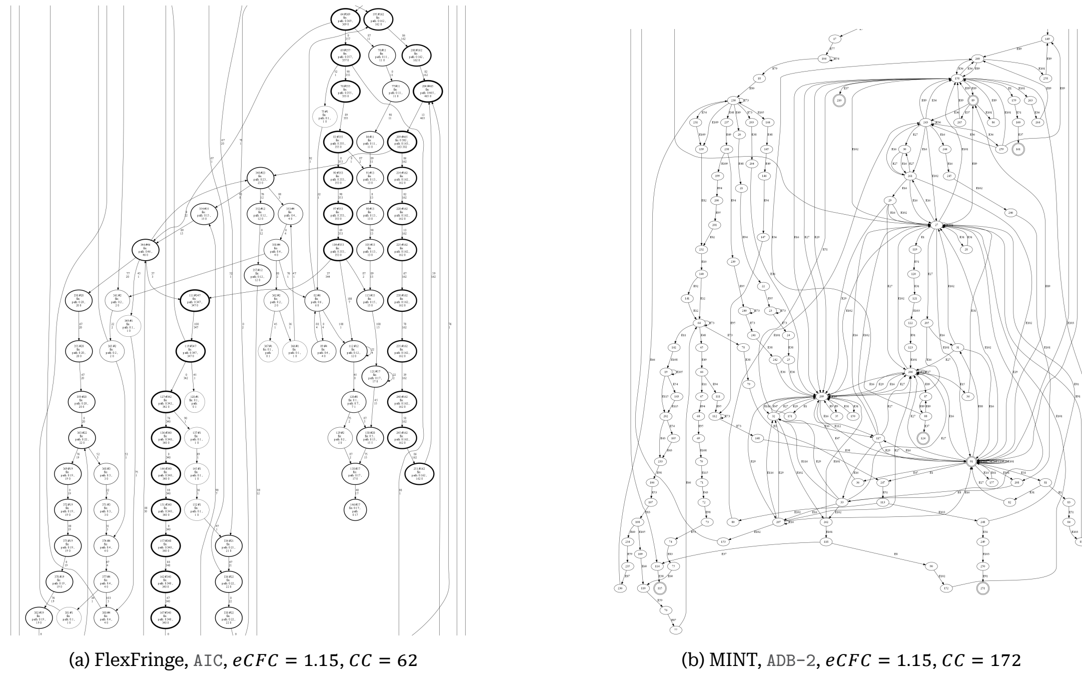
3. *Converting metrics to an equivalent metric*. This was tried for the TBR fitness, as explained in section 4.4.6. Indeed, the results proved not to be very informative and possibly not even useful.

This section will go into the troublesome metrics and their flaws.

6.1.1 Complexity

The complexity comparison seemed straightforward at first, but it did raise some issues. First of all, an FSM that scored better than others in terms of $eCFC$ could be scoring *much* worse in terms

of the CC than others. In addition to this, is the CC useful in any way for Petri nets? Does it express complexity when they are compared to each other or to the FSMs? Lastly, should a Petri net with many parallel structures be scored differently?



(a) FlexFringe, AIC, $eCFC$ = 1.15, CC = 62

(b) MINT, ADB-2, $eCFC$ = 1.15, CC = 172

Figure 6.1: PDApp fold 5, partial models

Does the $eCFC$ adequately show complexity in state machines?

Appendix E shows Spark models with their CC and $eCFC$ can be seen for each configuration. For the Spark data set, the scores feel intuitive on visual inspection. Indeed, the HDFS examples in figures 6.3 and 6.4 also express the visual inferiority of the ADB-2 model to the IM model. However, some $eCFC$ values for FSMs are equal while their CC is very different. An example of this occurrence is PDApp for the AIC and ADB-2 configurations. Figure 6.1 shows the most complex portion for both of these models. On manual inspection, the AIC model is indeed more readable. However, these models have the same $eCFC$ score, indicating that this score is not very accurate when comparing FSMs to each other. In addition to this, the lowest $eCFC$ one can achieve is 1, meaning these two models have achieve quite a low $eCFC$ score, in spite of being complex models. This is not unique to the PDApp set, there are many FSMs with very large CC score, that still perform well in terms of $eCFC$. This is because the $eCFC$ for FSMs is simply the ratio of edges to states. For example, the CoreSync models of AIC have a better CC than the models inferred by ADB-2. The models ADB-2 creates are extremely large and the discrepancy between states and edges is very large. AIC infers a more orderly model, as number of states and edges are smaller and the discrepancy is as well. Yet, their $eCFC$ is much better for the ADB-2 model as the ratio between edges and states is lower. In conclusion, the $eCFC$ has issues representing the complexity in a way that enables comparison of different FSM models and does not accurately represent the reality of the FSM complexity.

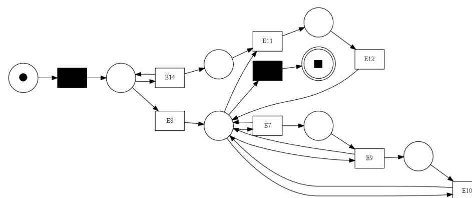


Figure 6.2: HDFS, fold 1, ProM Hybrid-ILP, $eCFC$ = 2.82, CC = 9

Was the CC useful for comparing Petri nets to FSMs, or even each other?

The CC works partially in terms of comparing Petri nets *to each other*. This was to be expected as the computation of the CC still adequately expresses how many 'extra' arcs go through the model. However, the CC is meant to express the amount of independent *paths* through the model and parallel constructs can greatly increase the amount of possible paths in a model. The CC is computed with the amount of arcs, transitions and places, but these counts do not express parallelism at all. This is the reason some Petri models with a high $eCFC$ still boast a low CC. An example is the HDFS `Hybrid-ILP` model, figure 6.2. It has multiple parallel constructs, resulting in OR-splits, but it has the lowest CC of them all. Comparing it to 2 other HDFS models, figure 6.3 and 6.4, it is certainly the easiest one to look at. But try to follow the chain of events, and this small model gets very complicated. Thus, the CC for Petri nets appears to be purely cosmetic and not representative of the true complexity of a process model. In addition to this, the CC for Petri nets will be lower if the model has a lot of parallelism.

In the comparison between FSMs and Petri nets, should parallelism be punished more?

As it stands, neither the $eCFC$ nor the CC pays much attention to the parallel construct. This construct does play a role in the readability of a model. Keeping track of a trace in a model with a lot of parallelism is much harder, as one needs to remember where in each branch they have left off. A good example is the HDFS model made by the inductive miner `IM`, seen in figure 6.3. It branches from silent transitions to various depths of the model and contains nested parallel constructs. However, the model is easy to read, unlike the HDFS spaghetti-bowl by `ADB-2`, seen in figure 6.4.

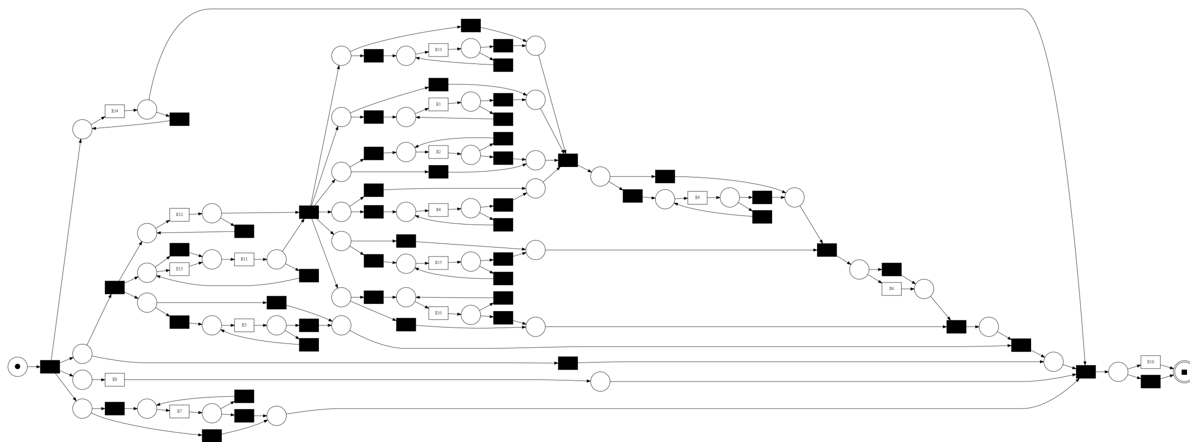


Figure 6.3: HDFS, fold 1, ProM `IM`, $eCFC = 1.23$, $CC = 36$

The answer to this question depends on what is valued the most. If it is purely visual readability, the CC complexity score reflect this well. Expressing parallelism and ease of manual tracing through the model would require a different metric or the $eCFC$ could be adjusted to add more than 1 for a parallel split. However, the issue of comparing FSMs to Petri nets would remain, as the $eCFC$ does not reflect complexity of FSMs well.

6.1.2 Trace Fitness

More in depth trace analysis was done with TBR fitness f_{TBR}, percentage of traversed edges $\%trav$ and f_{AL}. This turned out to be a worth wile additions, because e.g. the `AIC` model of Linux performs very differently in terms of perplexity and f_{AL} than its 0.0 recall would suggest. In this particular example, this was caused by transitions in test set that were not present in the training set. The f_{AL} did not cause any problems as it could be run on the converted FSMs without issues. The only limiting factor was the time and space required for the alignments. The f_{TBR} was problematic, which is something than can be analysed by comparing it to $\%trav$.

The f_{TBR} only reported 3 values under the 0.9 mark for the FSMs. This may not seem surprising, as the f_{AL} is also very high. However, there are two indicators that this fitness does not work for FSMs. First of all, there is the issue described in section 4.4.6, where the lack of the concept of tokens and Petri net constructs results in the fitness for an invalid trace becoming $f_{TBR,FSM}(t_{invalid}) =$

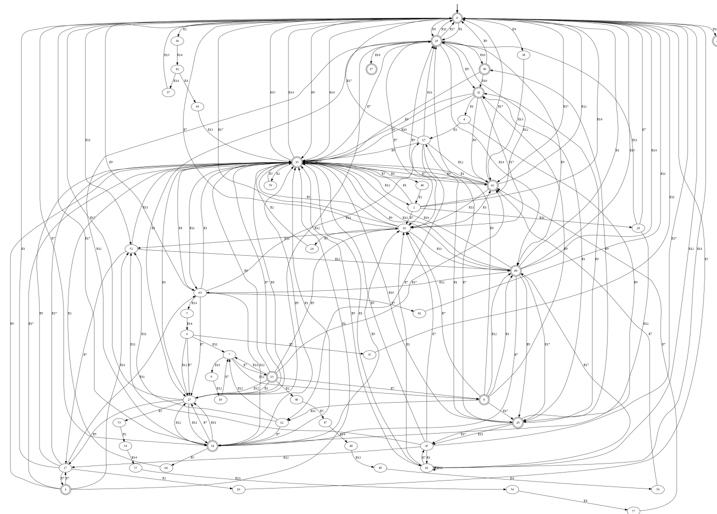


Figure 6.4: HDFS, fold 1, MINT ADB-2, $eCFC = 3.52$, $CC = 138$

$1 - \frac{1}{p}$. In spite of the high f_{AL}, one can still see that this definitely happened, with the help of the $\%trav$. The $1 - \frac{1}{p}$ relation tells us that (1) only smaller traces can have a lower f_{TBR} and (2) a long trace with more than 20 compliant events, will still get a high f_{TBR}.

Of the sets with smaller traces, only AIC NGLClient reports an f_{TBR} below 0.9. This does not disprove (1) as one can see that the percentage of successfully travelled edges $\%trav$ is also very high. Although a low percentage does not say much about the fitness of a trace, a high percentage does mean most of the trace was successfully replayed. Thus indicating that a large portion of the model was correct for this trace.

The AIC for Linux clearly shows that (2) happened. Linux's traces are very long, resulting in a high fitness. The $\%trav$ is very low, indicating the trace replay stopped fairly early in the trace. The computation of f_{TBR} stops when it encounters an invalid trace, and thus there should not be such a large discrepancy between it and $\%trav$. It manages to dip below 0.9 because the amount of compliant traces until an invalid transition is encountered in the test trace is very low.

Interestingly, the DF does not have such problems using the $PM4Py$ implementation. Even though this model also does not utilize Petri net structures and only has 1 token in its net. This hints at a possibility of using the standard implementation, if the Petri net implementation was changed or the Petri net conversion could be done differently without changing the meaning of the FSM.

It is clear that the f_{TBR} cannot be used as is. If the implementation of it could get fixed for the silent and duplicate transitions, it could become a viable option. The advantage of f_{TBR} is that its computation is much less space and time consuming than f_{AL}. The TBR fitness cannot be supplemented with $\%trav$ as this does not give the extra information needed. Furthermore, the $\%trav$ does not add anything to the comparison that the recall metric does not as $\%trav$ also does not take the rest of the trace into account and does not express whether a trace ended anywhere near an accept state. The f_{AL} is more suited to analyse both Petri nets and FSMs as it does not rely on the notion of tokens, which are nonexistent in FSMs. The f_{AL} is a great addition to analysis, provided it manages to complete its alignments.

6.1.3 Perplexity

Perplexity was successfully computed for both FSMs and Petri nets. There are two differences between these model types that require attention: parallel constructs and silent transitions τ. The parallelism difference is solved by effectively making a partial reachability graph with only the markings that occur for the test traces. For the τ transition, two decisions were made:

- The probability assigned to τ is computed in the same way as the symbols for other probabilities. Another option would have been to assign a very small probability, as for the *unseen*

event. However, this penalises a valid path too much and results in comically high perplexities.

- τ transitions are not considered as the same symbol. This decision was taken as tau is technically *not* a symbol at all. In addition to this, if tau transitions are considered the same symbol, the perplexity of a model can become very low, in spite of the huge amount of paths that can be traced through the model.

These choices were made as they were the most logical. This problem is not specific to Petri net to FSM comparison; these choices are the same if one were to compare an NFA with ε transitions to a DFA. The validity of these choices could be investigated further by computing perplexities for an NFA model with ε transitions and its corresponding DFA. This would make it easier to judge whether the penalties lead to an appropriate perplexity for the NFA. However, it will still be a matter of choice and not a definitive answer.

Generally, the reported perplexity values seem in line with the amount of parallelism in the ProM models. `Hybrid-ILP` reports some perplexities close to those of the FSMs, even though its performance on recall and $\%trav$ is consistently low (except on Hadoop and Spark). This may be due to the models with a small CC it creates and/or the choice of the unseen event probability p_{unseen}. When the CC of a Petri net model is small, it means not many states and transitions have multiple outgoing arcs. This means the probabilities between markings will be high, resulting in a lower perplexity. Additionally, if the alphabet of a model is small, this means the p_{unseen} can be quite large in comparison to that of other models. For example, one of the `Hybrid-ILP` CoreSync models has an alphabet size of only 7. This may have resulted in a choice of p_{unseen} that is not high enough to properly penalise traces that cannot be replayed.

Still, the perplexity scores do all represent what they ought to: how perplexed a model is by a specific trace. The PP score of `Hybrid-ILP` may be too optimistic if its recall and other scores are considered. However, in CoreSync, for example, many traces are short and can be represented by the small `Hybrid-ILP` model, so its low scores are not completely undeserved. In conclusion, the models need additional metrics to judge them properly, but perplexity is a valuable addition that can be computed fast. The small computation time is a big advantage, as many other Petri net computations can take an extremely long time.

6.2 Correctness of Models

How well models were able to perform, was measured with soundness, recall, precision, specificity, F_2-score, BA, trace fitness and perplexity. This section will go into the causes for the differences in performance for specific metrics. In terms of general performance, BA and F_2-scores, the FSMs generally outperform the ProM configurations. `Hybrid-ILP` ranks at the bottom, while `ADB-2` consistently performs well.

6.2.1 Soundness

An important facet of model correctness for Petri nets is soundness. The only ProM configuration that performed badly in this regard, is `Hybrid-ILP`. This is because the `IM` and `IMfa` configurations guarantee soundness and `DF` models are also sound by design. Both PRINS configurations and `ADB-2` also produced 'unsound' models, as far as FSMs can be unsound. The dead state in a deterministic machine is a state that is not accepting, and has only outgoing transitions to itself. In terms of DFAs and FSMs, this does not make the machine invalid. The notion is more important for Petri nets as unsound models can, for example, introduce unbounded behaviour, thus creating an additional hindrance to Petri net analysis. An example can be seen in figure 6.5, where repetition of event 204 causes a buildup of tokens in the red place.

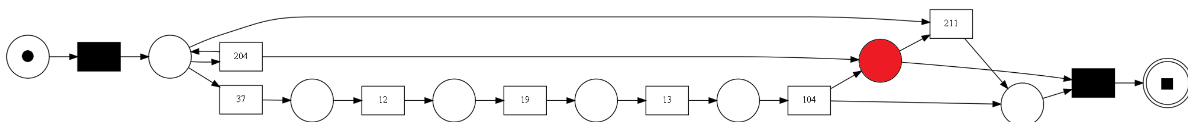


Figure 6.5: CoreSync, fold 1, ProM `Hybrid-ILP`, unsound; red state is unbounded

6.2.2 Overfitting and Generalisation in FSMs

Overfitting happens when a model fits only the training data and cannot handle new, slightly different data. To counter this, steps can be taken to *generalise* a model. This means the model allows more behaviour. Whether a models allows for too much behaviour, can be checked with negative traces and the specificity. A clue to possible overfitting is a model that has low recall on sets with traces that are not similar to each other and a high specificity. In addition to this, the model needs to have a good or perfect recall on the training data. This sub-section is geared towards the FSMs, as the ProM algorithms that were used, all have some 'overfitting' ingrained. They do not have the concept of state merging and only add directly-follows relations from the log. The only ProM configurations, `IM` and `IMfa`, that can generalise, only do this when they fail to find a cut. They do try to use a fall-trough that will not introduce too much behaviour, but can resort to using the flower model. Their recall and specificity results will be discussed in the next sub-section.

A configuration that possibly overfits is `AIC`. For all data sets, it has perfect recall on the training data (table C.1). Its recall on most data sets is good, except for Linux and Zookeeper. However, the specificity for these data sets is still satisfactory. This may be due to overfitting. Further investigation of the Linux and Zookeeper test and training data shows none of the traces in the test fold were exactly the same as the training traces. The third worst recall for `AIC` is on Oobelib, which introduces around 70% new traces, but this recall is still 0.79. The large performance difference may be due to a combination of factors in the data characteristics. In addition to introducing all new traces in the test set, Linux and Zookeeper also have a trace similarity of $\overline{S} < 0.25$ whereas Oobelib has a similarity of 0.66. Other sets with low similarity also have a low percentage of new traces introduced. It should be noted that a different trace does not necessarily mean new transitions or bigrams were introduced. From these results, it can be concluded that `AIC` overfits on the training data, and does not manage to generalise well if the traces are not similar and the test data consist of traces that are not seen during training.

`ADB-2` and the PRINS configurations do not seem to overfit. Their specificity fluctuates and is generally not high. The gap between the train and test data recall for Linux and Zookeeper is not large. Furthermore, these configurations have a much better recall than `AIC` on these two sets, indicating that they handle new traces better. This may be due to the classifier in MINT. Candidate pairs for state merging are checked with the classifier for consistency as well. This classifier expresses whether one event is likely to happen after the other event. Therefore, even if the trace was not seen during training, the bigrams of the trace may have. Another reason could be that the configurations overgeneralise and allow for too much behaviour, thus causing the low specificity. `W4-HD1` and `W4-NFA` perform similar on recall and specificity. `ADB-2` also has similar recall scores, but scores lower on specificity. This tells us three things. Firstly, the hybrid determinisation for `W4-HD1` retained most information and behaviour of its `W4-NFA` model. Secondly, PRINS' method of stitching `ADB-2` component models together results in a similar ability to recognise valid traces. Lastly, `ADB-2` generalises the model more than the PRINS configurations. The fact that PRINS generalises less than the `ADB-2` configuration, despite using it internally, is most likely caused by to how it stitches its component models together. PRINS uses MINT to make models for each component in the data set and then effectively creates a tree-like structure where each trace is a branch, as seen in figure 6.6. This structure is not entirely unlike a prefix tree, except for whole traces. This means in most cases it will be bigger than a prefix tree and fit most traces. Unlike a prefix tree, it does not have perfect recall on the training set. The way PRINS stitches, it ought to append each part of the component models corresponding to all partitions of a trace and thus achieve perfect recall. The fact that is does not, must be due to `ADB-2` component models that did perfectly fit to its component log. This can and does happen, as seen in table C.1, where the `ADB-2` training set recall is either similar to or a bit ahead of PRINS's.

Generalisation and Synthesised Traces

When analysing generalisation, one must keep the reliability of the synthesised negative traces in mind. Generalisation is effectively a question of how much additional, slightly different behaviour a model should allow. If the software logs are known to be complete, i.e. they contain *all* possible behaviour, generalisation is not necessary or desirable. Unfortunately, it is not known if the log are in fact complete, so the configurations must generalise properly to be able to recognise new, slightly different data. The negative traces were synthesised by making very small mutations until the result

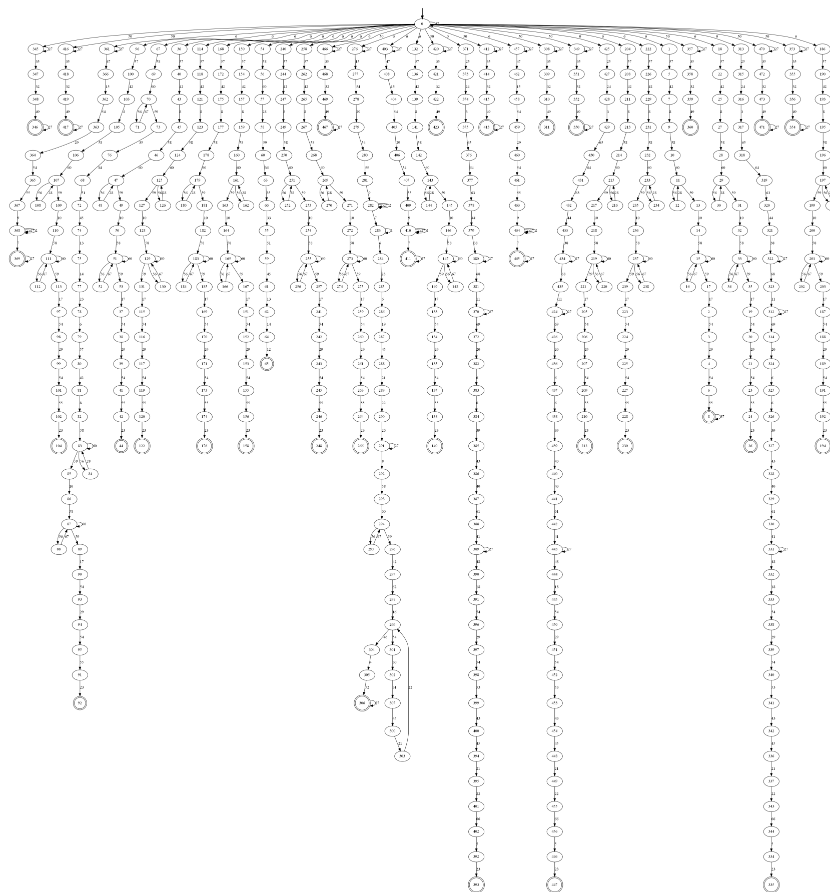


Figure 6.6: NGLClient, fold 5, PRINS W4-NFA

was a trace that was not in the full data set. It is not know with certainty that these sequences of events are in fact impossible in the software. Due to this, a model that generalised in such a way that it accepts negative traces with small mutations, cannot definitively be labelled as overgeneralised or 'bad'.

6.2.3 Recall and Specificity for ProM Models

The inductive miners do not consistently report good recall or specificity. The poor specificity scores for the IM miner can easily be explained. If the IM algorithm cannot find a cut, it will use one of the fall through. Each fall-though introduces some additional behaviour; IM tries to use the fall-throughs that add the least behaviour first. In the worst case scenario, the flower model is produced for a sub logs, which will allow any sequence of that sub-alphabet. That may explain why so much extra behaviour appears to be allowed in most IM models. The same can be said for the IMfa configuration, which has the same fall-throughs. Interestingly, IM reports better much recall than IMfa in some sets, whereas IMfa reports much higher specificity in some sets. This is due to the frequency filtering in IMfa. As explained previously, IM may suffer from not finding cuts and using a fall-through. If the frequency filtering is applied successfully, IMfa may find a cut where IM does not. This could also explain why IMfa has more problems with the Linux and Zookeeper data sets; IM probably used a fall-through that allows extra behaviour, or even the flower model, on some of the sub-traces.

Hybrid-ILP consistently reports low recall and a high specificity. This can be due to the filtering. On the one hand, the filtering may have been to aggressive. This may explain why Hybrid-ILP does not perform as well as others on Hadoop. On the other hand, many of the logs have dissimilar traces and are noisy. This can lead to overly restrictive constraints for the ILP. If the filtering worked well, it should solve this. But from these poor results, it can be concluded that Hybrid-ILP cannot deal

with noisy logs or with new traces.

The DF configuration has a stable performance overall. The threshold of 0.8 effectively a guarantees that at least 80% of the training data will fit on the model [25]. The specificity scores for this configuration are fairly similar to its recall scores. Its scores relatively low on Hadoop, compared to the other configurations. Upon closer inspection, DF only performs a bit worse on 2 of the 5 folds (recall = 0.692). These are exactly the same folds that have a non-zero percentage of traces in the test set that are not in the training set. Indeed, the configuration also has trouble with the other sets that introduce many new traces in the test data: Linux and Zookeeper. This is not surprising; DF creates a directly follows graph for the data and then filters the traces to create a new graph. This leads to adequate performance when a log is (almost) complete, but it will struggle to recognise new traces.

Performance Deviation Across Folds

There is a noticeable difference in the error bars of the FSM models and the ProM models. The ProM models have more deviation across folds, except for IM and IMfa on Hadoop. The Hadoop set has a trace similarity of $\overline{S} = 0.98$ across all traces, so all traces are almost identical. This is most likely due to the aforementioned characteristic of all used ProM models: they only add directly-follows relations they see in the training data. The models made for each fold can differ a lot from each other depending on which traces were seen. If there are many infrequent traces, the filters for IMfa, DF and Hybrid-ILP.

6.2.4 Precision

The tendency to classify traces as positive was also measured with the precision. A lower score means a configuration tends to classify a lot of traces as positive that are in fact negative. AIC generally does not have the tendency to recognise too many traces as positive. It only performs poorly on this metric on Zookeeper and Linux. This is not due to it reporting a lot of false positives, but due to it recognising few true positives for these sets. The reason for this was already discussed in the previous sub-sections. ADB-2 also performs much worse on Zookeeper and Linux, but for the opposite reason: it has more false positives. The PRINS configurations outperform ADB-2 for all sets because it reports less false positives. DF reports similar values across recall, precision and specificity because the amounts of misclassified traces are similar for positive and negative traces. This also happens for both Inductive miners on some sets. Hybrid-ILP often reports good precision as it has a low amount of false positives and a low amount of true positives.

6.2.5 Alignment Fitness

Unfortunately, the alignment fitness took a long time to compute. This seems strange, as the alignment fitness for some of the sets that took a while to compute was near 1. And if a trace almost aligns, one would expect it to compute faster. Some computations, especially those of ProM models, terminated with memory errors as well. This resulted in partial computations for some sets or no results at all. This can have a big impact on the average score. For example, AIC f_{AL} for Linux is near 1, but only 7.5% of alignments were computed. Only 3 of the 8 traces were aligned, for one fold only. This is not a reliable result and cannot be used for analysis. All alignments with less than 75% completion will not be discussed in this sub-section.

Although the ProM models did not always perform well on the general metrics, some did well on alignments. For IM and IMfa satisfactory alignment completion was reached for Coresync, Hadoop, HDFS, NGLClient and Spark. It performed excellently on these sets and did not report high alignment costs, thus the reason for the low recalls on CoreSync, HDFS and NGLClient was due to minor deviations of the test log from the model. DF completed almost all alignments, except for Zookeeper's. It performs very well, its lowest f_{AL} are around 0.85 for CoreSync, NGLClient and PDApp. Its alignment cost is always among the highest. Hybrid-ILP also completed all of its alignments except Zookeeper. It reports score over 0.8 for Hadoop, HDFS, Linux and Spark. Apparently, the deviations of the logs to the model on other sets were large, with the f_{AL} on some of CoreSync's fold even falling below 0.3. Which makes sense, as many of its CoreSync models can only recognise one trace with 5 events. Unsurprisingly, it also reports the higher alignment cost on almost all sets.

AIC did not complete its alignments for Linux, Oobelib and Zookeeper. Unfortunately, these were also the sets with lower recall, so it would have been interesting to know their alignment fitness. Recall and $\%trav$ is near 1 for the other sets, so it is not surprising that their alignments are near 1 as well. Only NGLClient reported an alignment around 0.8 on two folds, with a higher alignment cost. The PRINS configurations did not manage to complete many alignments. For W4-NFA, this was most likely due to memory errors because the huge models it produced. Only Hadoop and NGLClient completed for W4-NFA, and this is because the recall was 1 for these sets. W4-HD1 completed Hadoop, NGLClient, HDFS and Spark. The only lower fitness values it reports was on the same fold of NGLClient as AIC, but no obvious difference between that fold and the other ones could be found. ADB-2 did not complete its alignments for CoreSync and Zookeeper. Again, all alignments that were computed are near 1 and the alignment costs are low.

Although alignment fitness is a suitable metric, it would appear that the most interesting alignments could not be computed within the time limit. This is a problem for proper analysis, as all FSM results that were completed were obvious by looking at their high recall and $\%trav$.

6.2.6 Perplexity

The perplexity reported is generally highest for the inductive miners. This makes sense, as the models have many parallel structures and are much larger than Hybrid-ILP. In fact, all data sets where the IM and IMfa perplexities are $PP > 15$ are also the only Petri net models with over a 100 states and transitions.

The influence of the choice of p_{unseen} can be seen in the AIC perplexities for Zookeeper and Linux. These perplexities are the highest peaks for AIC, which is due to the penalty given when a trace does not end in a sink. On the other hand, the low perplexities for Hybrid-ILP do not necessarily reflect good models, but are likely caused by the extremely small models it makes due to filtering or its ILP constraints. Hybrid-ILP models for CoreSync and HDFS have fewer than 20 places and transitions, a low cyclomatic number and only 2-3 transitions causing parallelism. If there is fewer possible transitions from a markings, the probability for each transition will be higher and the perplexity lower. Since not many traces get to the end in the Hybrid-ILP models, a lower p_{unseen} may lead to more representative perplexities. It is difficult to make this judgement for IM and IMfa, since the model size and amount of parallelism makes it impossible to approximate the size of its transition graph.

The PP for IM and IMfa configurations is affected a lot by the choice to not consider τ transitions as an equal symbol. However, not doing this can result in very low perplexities as many of the τ transitions introduce parallelism, as can be seen in figure 6.3. Therefore, considering them as an equal symbol will often result in it receiving a probability of 1.

It is interesting that AIC beats the PRINS and ADB-2 configurations, as it does not always perform better in other metrics. In fact, even for the data sets with the worst recall, AIC outperforms the other FSM configurations. Perplexity scores for DF spike on the data sets where it reports a cyclomatic number over 100, whereas the other data sets have a $CC < 30$.

6.2.7 Conclusion

The configurations performed well on different metrics. To conclude this section, each performance metric discussed so far will be listed, with its top performers:

- **Soundness**: all configurations did well, except for Hybrid-ILP, which only made about 30% sound models.
- **Recall**: Overall recall is the highest for ADB-2 and the PRINS configurations. If Linux and Zookeeper are not considered, AIC performs excellent as well. Much like AIC, the IM and DF only perform poorly on 2 data sets.
- **Specificity**: AIC and Hybrid-ILP. All other configurations, except IM, are also > 0.7.
- **Precision**: The top 6 configurations are all close together. The bottom two are IMfa and IM.
- **Balanced Accuracy** BA: the PRINS configurations, ADB-2 and AIC.
- F_2**-score**: ADB-2 and the PRINS configurations.

- **Generalisation**: ADB-2.
- **Alignment fitness** f_{AL}: Everyone is a winner, except for Hybrid-ILP. Many alignments did not complete though, and it is possible that these were the low fitness values.
- **Perplexity** PP: AIC. The worst are Hybrid-ILP, IM and IMfa.

6.3 Model Complexity and Performance

Undoubtedly, Hybrid-ILP is the king of small models. But, there is a problem. Hybrid-ILP performs much worse in almost every other way. In fact, the trade-off is most certainly not worth it. This is also true for IM and IMfa. With the exception of Hadoop, HDFS and Spark, they create large Petri nets with a high CC. In reality, these models will be even harder to read due to the parallel structures. They do not perform consistently well on other metrics, except for the alignment fitness, although their alignments were not fully completed.

DF has no parallelism, and the lowest overall CC. It performs neither the best nor the worst on anything. Due to its filtering method, it does not really generalise. W4-NFA is meant to be determinised and was added as to compare it to W4-HD1 and ensure PRINS would return results even if determinisation took too long. It is obvious from its number states, transitions and CC, that this should never be used over W4-HD1 and needs determinisation. However, since so little information is lost compared to ADB-2, it may still be interesting to create a different determinisation for it.

As for using W4-HD1 over ADB-2, in terms of complexity this is not an attractive option. It manages to makes a smaller of similar sized models for Hadoop, NGLClient and Spark, but for the other sets, the amount of states can balloon to 10 times as high as the ADB-2 state count. The small performance increase on some metrics is not worth it. This may also explain why W4-HD1 outperforms ADB-2 sometimes; a larger model has more room to model a lot of very specific behaviour. In terms of complexity and performance, ADB-2 is the better choice.

AIC models generally have less states than ADB-2, and a lower CC as well. ADB-2 is much better in terms of complexity on Zookeeper and Linux, which are also the sets that AIC performs badly on. The ADB-2 Spark model is also smaller, but the AIC model is not large either. ADB-2 performs better in every way, except on perplexity and specificity. Overall, ADB-2 models are 1.2 times larger. However, its peaks extreme: the CoreSync model has on average 3958 states and a $CC = 838$. It is 8 times as big as the AIC model. The ADB-2 model for PDApp is also 2.5 times larger.

For data sets that are not like Linux and Zookeeper, AIC and DF are the better choices in terms of complexity and performance. However, ADB-2 has good and steady performance across all metrics. Its models are generally small, but for CoreSync and PDApp its complexity becomes extreme, whereas AIC does not have such massive peaks.

6.4 Inference Time and Performance

All ProM configurations and AIC were extremely speedy if inference was completed. Hybrid-ILP did not create all models for Oobelib and Zookeeper. PRINS and ADB-2 could also not complete inference on Oobelib. The determinisation for W4-HD1 also timed out for PDApp. The inference for Spark was incomplete for some configurations, but this was due to memory errors, not a timeout.

AIC runs fast and outperforms the ProM configurations. The question that remains is whether the performance of PRINS and ADB-2 is worth the extra time inference takes. On average, inference time for W4-NFA was around 20 times that of AIC, W4-HD1 40 times and ADB-2 is 120 times as slow. PRINS was created to mitigate the long inference time ADB-2, and it succeeds at that. Improving the hybrid determinisation run time can greatly increase the run time of PRINS. The performance of W4-HD1 is on par with that of ADB-2, so in terms of run time, W4-HD1 is the better choice. However, AIC is so much faster, while making smaller models, that in some cases the performance decrease may be worth it. DF was also extremely fast and has a steady performance for most sets.

6.5 Data Set Characteristics Influence

The analysis that was done for the influence of data set characteristics (trace similarities, normalised event entropy and unique events) on performance did not yield any definitive results. Although FSM inference appears to be more stable when the trace similarity is higher, there is no

definitive proof that one metric in fact causes the other. To make matters worse, there are not enough data points for certain ranges of characteristics values.

There can be two different reasons for the lack of discernible shapes in the graphs. Firstly, there may just not be a correlation between any of the metrics that were investigated. The other cause could be the data sets themselves. The data sets are diverse in many different ways. It is possible that there are too many variables involved to do an analysis targeted at one characteristic. Possibly, it is necessary to have data sets where all variables are known and/or can be controlled.

6.6 Limitations

This section goes into the problems and limitations that were encountered during this research. The over-arcing theme is that the research was more broad than initially thought, thus some elements could not be investigated as much as desired or were entirely left out.

6.6.1 Parameter Tweaking

Several configurations had parameters to choose: the noise threshold for `IMfa`, the filters for `Hybrid-ILP`, the noise filter for `DF`, the merge score in `ADB-2` and PRINS' HD_u parameter. And these are only the parameters of the methods. For example, FlexFringe also implements other algorithms besides `AIC` and MINT can be run with a different classifier instead of AdaBoost. These parameters can make a difference in performance for certain sets. For example, `DF` filters noise in the Hadoop data set, but this data does not contain noise and does not need filtering. If the threshold had been properly tweaked, its performance may be on par with the rest of the configurations. Another example is the choice of AdaBoost and its merge parameter. This configuration was chosen as it had the best performance *overall* in the comparison by Walkinshaw et al. [53]. However, it was outperformed on some of their data sets by other parameter values.

6.6.2 Time and Memory

Some of the models and measurements on the models could not complete due to the time constraints. This work created models for 5 data folds on 9 data sets, which limited the time that could be spent on each model inference and each metric measurement. Most of the measurement computation time went into the alignments and the soundness.

Additionally, both inference and measurements suffered from memory errors due to too little RAM. All computations were done on a laptop with `16GB`, but this was not enough for MINT inference, alignments and soundness computations. For much of the time that inference and measurements were running, the laptop was not able to be used by any other processes.

6.6.3 ProM CLI

It was not feasible to manually run each fold for each data set in ProM's graphical user interface. Aside from how much trouble it would be to turn on a miner 180 times, the user interface is prone to freezes if a large computation is run. To avoid this, ProM was run from its Command Line Interface. ProM CLI has a long startup time, so all mining was done in one run. This was a problem as miners needed a timeout. The script to run the miners was in Java, but the inner workings of ProM code do not always allow for a safe and reliable timeout. However, this all paled in comparison to the biggest issue. There is not much documentation and it was difficult to find out how to call the desired miner and set its parameters properly.

6.6.4 Data Sets

As discussed earlier, the lack of reliable negative traces holds proper investigation of generalisation back. Any further research would benefit greatly from a data set with verifiable negative traces, even if they are not used for training. For the purposes of investigating data set characteristics, it is necessary to have sets that are more uniform on several characteristics. With these varied sets, it is impossible to know if one characteristic truly influence anything, if a bunch of other characteristics differ a lot as well. However, for the purposes of a general analysis of performance, the variety was very useful.

7 Conclusion

This thesis set out to answer the question of how one would choose between ProM process miners, FlexFringe, MINT and PRINS to evaluate software traces. To choose between these tools, it is necessary to be able to compare Petri nets with FSMs. This comparison proved more difficult then expected. The parallel structures in Petri nets made for high computation times and high memory usage during mining and analysis. Nonetheless, suitable performance metrics were identified for Petri nets and FSMs. Analysis with these metrics did not yield an overall winner; the choice depends on the needs of the user and on the data.

How can Petri nets and automata output be compared?

At first glance, Petri nets and FSMs can easily be compared in terms of run times and classic metrics like recall, specificity and F-score. However, issues arose when comparing model complexity and when trying to do a deeper analysis of trace fitness. This work did not identify a suitable complexity metric for both Petri nets and FSMs that truly expresses the number of paths through a model. The biggest disconnect in comparing complexity of a Petri net and an FSM is caused by the parallel structures of a Petri net. This structure can express many different paths, yet look visually pleasing. However, if one were to follow a trace through a model, this structure can be more complicated than a spaghetti-like model. Neither the $eCFC$ nor the CC punishes parallel structures and the $eCFC$ does not express complexity in FSMs well.

The trace fitness could be compared by transforming the FSMs into Petri nets and using Token-based replay and alignment-based fitness. The TBR fitness did not work well, as it relies to much on the notion of tokens, which do not exist in FSMs. Furthermore, the implementation of the algorithm did not work properly for the converted FSMs as it had problems finding a path. Alignment-based fitness works with a converted FSM and does not use any constructs that are not in FSMs. The alignment-based fitness analysis was only held back by the large amount of time and memory it took to complete the alignments. The computations were incomplete for the larger models (PRINS, MINT, Inductive Miner).

The Perplexity metric was computed on Petri nets by following a trace and using the travelled markings to determine the amount of outward transitions in its reachability graph. Choosing silent transition τ to always have probability of $\frac{1}{|N|}$, where $|N|$ is the total number of outwards transitions of a marking, yielded results that were in line with the amount of parallel structures in the Petri nets.

Can the programs produce a model in feasible time and memory?

FlexFringe, ProM's Inductive Miners and the Directly Follows miner all produce results in a small amount of time. They did not time-out or run out of memory on any of the data sets. Moreover, the highest inference time for these configurations was 40 seconds. These configurations do not report the best overall performance, but for data sets with over $20,000$ log entries, they are significantly faster than PRINS and MINT. Furthermore, the PRINS and MINT timed out on Oobelib and only returned partial results for Spark due to running out of memory. `Hybrid-ILP` also timed out on Oobelib, and returned only one out of five models for Zookeeper.

How accurate and correct are the produced models?

First of all, `Hybrid-ILP` did not do well in terms of soundness. It mined 36 models, and only 11 of them were sound. FSMs do not have a real notion of soundness, but it was computed anyway. This tells us that FlexFringe does not make models with dead states and few of the MINT models have dead states.

MINT and PRINS are excellent at recognising the true positives. FlexFringe and the Inductive Miner are as well, but both fail on two of the data sets. For FlexFringe, this is due to it producing a perfect fit to the training data, and not generalising much. When a new data is introduced and the traces are not similar, FlexFringe has issues recognising true positives. This also happens for the Inductive Miner, but its fall-throughs can allow extra behaviour, enabling the miner to achieve a better recall. The Inductive Miner variant `IMfa` was outperformed in terms of recall by the base miner on some sets as it utilises less fall-throughs due to filtering. The Directly Follows miner failed for the same sets as FlexFringe: Zookeeper and Linux. This was also due to it not modelling any new behaviour. For the ProM miners, this is by design. ProM miners apply frequency filtering to reduce the model size and complexity, and generalisation in Inductive Miners is due to fall-throughs. This is different for state merging algorithms, which can introduce new behaviour due to the choice of merge scoring.

True negatives and false positives were measured using synthesised negative traces. These traces have small mutations and could not be verified as truly negative. This interfered with the reliability of the specificity and precision, and by extension the balanced accuracy and F_2-score. Although all configurations reported adequate specificity, there is no decisive results on how good models are at rejecting negative traces. The specificity for Inductive Miner does align with intuition: it can achieve poor specificity due to its fall-through structures allowing too much behaviour. From the results that were reported, the ProM and FlexFringe configurations fit to the training data, and MINT and PRINS generalise more. The best balanced accuracy and F_2-scores were achieved by MINT and PRINS.

Most alignments that did complete, were the ones that reported a high fitness. The only lower fitness values were the few sound `Hybrid-ILP` models. The computation was successful because these models were very small. Intuitively, it takes more time and memory to compute an alignment on a trace that does not align well, so it is possible that some of the fitness values that would differentiate the configurations more are missing. As it stands now, all configurations achieve excellent fitness, except `Hybrid-ILP`.

Perplexities were all computed successfully, in a short amount of time. Due to the parallelism in the Petri nets, they generally report worse scores, with the exception of the Directly Follows miner as it does not model parallelism. FlexFringe reports the best perplexities and beats MINT and PRINS. The perplexity of `Hybrid-ILP` models are deceptively low for its small models with little parallelism. This may be due to the probability of unseen events being too high, thus not penalising traces that do not finish in an end place enough. The MINT, PRINS and Directly Follows miner all reported good perplexities. The perplexities of the Inductive Miners spike for models with a lot of parallelism.

How complex are the models?

Although complexity was difficult to compare, Petri nets with parallelism that were near the FSMs in size, must be much more complicated. More than half of the Inductive Miner's models are equal to or larger in size than the FlexFringe models. The results of the Directly Follows miner are easier to compare, as there is no parallelism. This miner makes the smallest models with the lowest CC overall. FlexFringe generally outperforms PRINS and MINT by quite a margin. PRINS models are so much larger than MINT models, one has to wonder if it is worth it. The computation time was a bit lower for PRINS, but in the time frame given, it timed out on the same amount of models as MINT. In addition to this, it does not outperform MINT by enough to justify such large models. MINT has extreme peaks on a few data sets in terms of states and transitions compared to FlexFringe and the Directly Follows miner.

Do data set characteristics have influence on performance?

No definitive proof of a correlation between any of the investigated data set characteristics and the performance for each configuration could be found. This could either be due to a lack of correlation

or because the data sets themselves are not suitable for such an analysis as there are too many variables in their characteristics.

Which Configuration Should Be Chosen?

The answer to the main question of this work is not simple. It depends on the data, the purpose of the model and the hardware available. If the data set is known not to be complete and new behaviour must be modeled, PRINS and MINT are more suitable. However, this comes at the cost of very long inference time and larger models. Therefore, if the data set contains many log entries, it may be wise to choose FlexFringe or the Inductive Miner. The `IMfa` should be chosen over the base miner if a data set is both (almost) complete and contains much infrequent behaviour. The Directly Follows is also a good contender as it is fast, creates small models and has overall good performance. However, this miner cannot truly introduce new behaviour as its output is purely based on the directly-follows relations in the data.

Time and memory are a big factor. This is where MINT and PRINS show some flaws. FlexFringe and all ProM configurations (save for `Hybrid-ILP`), had no issues with time-outs or running out of memory. MINT and PRINS returned with memory errors on one data set and ran out of time on another. If time, space and model complexity are important, FlexFringe and the Directly Follows miner are more suited to the task. They are fast, produce small models, have the best overall perplexity and give a good overall performance. The only exception for this were the 2 data sets were all traces had low similarities and the test sets introduced many new traces.

As for complexity and readability, FlexFringe and the Directly Follows miner are the right choice. The smaller `Hybrid-ILP` models do not work well and the Inductive Miners make relatively large models, with a lot of parallelism. Furthermore, if one would like to analyse their model further, computation times will be large for any model with parallelism. However, if the main concern is a visually appealing and ordered model, the Inductive Miners do achieve that.

PRINS is a method to speed up inference methods like MINT, while still providing the perks of MINT. It retains the excellent performance of MINT, but the models it creates are generally much larger and inference time can still be quite long. However, the PRINS configuration without determinisation shows that perhaps a change in how determinisation and state merging is performed, will mitigate the model size problem.

In conclusion, PRINS and MINT perform very well across all correctness metrics. They are more suited to recognise new behaviour than the ProM models. However, the inference takes a lot of time and space and can result in large models. FlexFringe and the Directly Follows miner provide a good trade-off for time and performance, but do not perform well when logs are incomplete and dissimilar to a high degree. Because most models mined by the `Hybrid-ILP` miner were not sound, using this miner is not advisable. The Inductive Miners are suitable for more complete software logs, but can produce very large models for logs with many events. In addition to this, the user needs to decide whether parallelism is acceptable and desirable for their use case.

Future Work

This research was meant as an exploratory work into the comparison of FSMs and Petri nets and how well they perform on real-life data. Certain aspects relating to the comparison methods were more complicated then expected or not possible with the used data sets and amount of inference methods. The parallelism in Petri nets is what caused most problems in terms of comparison and unfinished computation. Petri net analysis would greatly benefit from more research in to faster algorithms to compute or approximate metrics such as alignments.

Complexity Metrics

The current complexity metrics can only be used as a purely cosmetic indicator of how confusing the FSM and Petri net models look. The metric is superficial and does not represent true complexity in terms the amount of paths a model introduces. This is mainly caused by the parallelism in Petri nets. Although these parallel models are more visually pleasing, tracing paths through them is difficult, especially when parallel structures are nested. Parallelism causes high space and time complexities on Petri net problems, so a metric that expresses this structure properly could be beneficial. The amount of transitions inside a parallel structure influences the amount of paths that are

introduced. The paths can be expressed through a reachability graph, but the computation is often infeasible. Solving this without creating a reachability graph may prove challenging.

Token-Based Replay Fitness

This metric might work, if the implementation did not run into problems. The problem appeared to be the silent transitions connecting the start and accept states the the Petri net start and end place and the many duplicate transitions. It would be necessary to look deeper into why the converted FSMs could not be plugged into PM4Py's TBR function to fix the bug. Another option would be to find a different way to create an equivalent Petri net. Even if this problem is fixed, it may still be possible that the TBR fitness does not work well for the reasons described in section 4.4.6.

Perplexity

The perplexity was calculated with three assumptions: the unseen probability is $\frac{1}{alphabet\ size}$, the τ probability is computed as other symbols and multiple τ transitions are not considered the same symbol. This is a problem that is also applicable to a comparison of a DFA to an NFA with ϵ transitions. It would be beneficial to look into these choices further. Since each NFA can be translated to an equivalent DFA, it would be much easier to do such research on two equivalent machines instead of the many different models that were generated for this work.

Data Set Characteristics as Performance Predictor

The data in this work was too varied to provide a definitive answer to the last research objective. It ought to be investigated with data sets where only one characteristic changes, or the effect of the change on other characteristics is know. If there is a correlation between data set characteristics and performance, it may help decide which tool to choose for the data set.

Compare FSMs to Probabilistic Miners

There was a large disconnect between the intention of state merging in FSMs and filtering in the process miners. The intention of the used process miners appears to be to achieve a perfect fit to the training data. They do not intentionally generalise to recognise new behaviour. However, there are miners designed to be more suitable for incomplete software logs. One example is the Inductive miner for incomplete logs. This may make for a more interesting comparison in terms of generalisation.

PRINS determinisation

Comparing `ADB-2` with the two PRINS configurations, the performance is on par. It does compute either in a similar time or faster than `ADB-2`. The big drawback is that it creates extremely large models. This may be mitigated by applying a different determinisation or state merging method.

Bibliography

[1] Aalst, W. v. d. (2016). *Process mining: Data science in action*. Springer Publishing Company, Incorporated. https://doi.org/10.1007/978-3-662-49851-4

[2] Berti, A., & Aalst, W. M. P. v. d. (2019). Reviving token-based replay: Increasing speed while improving diagnostics.

[3] Berti, A., van Zelst, S. J., & van der Aalst, W. (2019). Process mining for python (pm4py): Bridging the gap between process-and data science. *arXiv preprint arXiv:1905.06169*. https://doi.org/10.48550/arXiv.1905.06169

[4] Biermann, A. W., & Feldman, J. A. (1972). On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, *21*(6), 592–597. https://doi.org/10.1109/TC.1972.5009015

[5] Bishop, C. M. (2006). *Pattern recognition and machine learning (information science and statistics)*. Springer-Verlag. https://doi.org/10.5555/1162264

[6] Cardoso, J. (2008). Business process control-flow complexity: Metric, evaluation, and validation. *International Journal of Web Services Research*, *5*(2), 49–76. https://doi.org/10.4018/jwsr.2008040103

[7] Cheng, K. T., & Krishnakumar, A. S. (1993). Automatic functional test generation using the extended finite state machine model. https://doi.org/10.1145/157485.164585

[8] Cohen, H., & Maoz, S. (2015). Have we seen enough traces? (t). *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 93–103. https://doi.org/10.1109/ASE.2015.62

[9] Cohen, W. W. (1995). Repeated incremental pruning to produce error reduction. *Machine Learning Proceedings of the Twelfth International Conference ML95*.

[10] Cook, J. E., & Wolf, A. L. (1998). Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol.*, *7*(3), 215–249. https://doi.org/10.1145/287000.287001

[11] Damas, C., Lambeau, B., Dupont, P., & van Lamsweerde, A. (2005). Generating annotated behavior models from end-user scenarios. *IEEE Transactions on Software Engineering*, *31*(12), 1056–1073. https://doi.org/10.1109/TSE.2005.138

[12] de la Higuera, C. (2005). A bibliographical study of grammatical inference. *Pattern Recogn.*, *38*(9), 1332–1348. https://doi.org/10.1016/j.patcog.2005.01.003

[13] De Weerdt, J., De Backer, M., Vanthienen, J., & Baesens, B. (2012). A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. *Information Systems*, *37*(7), 654–676. https://doi.org/10.1016/j.is.2012.02.004

[14] Emam, S. S., & Miller, J. (2018). Inferring extended probabilistic finite-state automaton models from software executions. *ACM Trans. Softw. Eng. Methodol.*, *27*(1), Article 4. https://doi.org/10.1145/3196883

[15] Esparza, J. (1998). Reachability in live and safe free-choice petri nets is np-complete. *Theoretical Computer Science*, *198*(1), 211–224. https://doi.org/https://doi.org/10.1016/S0304-3975(97)00235-1

[16] Freund, Y., & Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, *55*(1), 119–139. https://doi.org/https://doi.org/10.1006/jcss.1997.1504

[17] Goutte, C., & Gaussier, E. (2005). A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. *European conference on information retrieval*, 345–359.

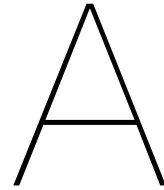
[18] Habben-Jansen, G. (2021). *Mining attack strategy* (Thesis).

[19] He, P., Zhu, J., Zheng, Z., & Lyu, M. R. (2017). Drain: An online log parsing approach with fixed depth tree. *2017 IEEE international conference on web services (ICWS)*, 33–40. https://doi.org/10.1109/ICWS.2017.13

[20] He, S., Zhu, J., He, P., & Lyu, M. (2020). *Loghub: A large collection of system log datasets towards automated log analytics*. https://doi.org/10.48550/arXiv.2008.06448

[21] Jurafsky, D., & Martin, J. H. (2000). Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition. Prentice Hall PTR. https://doi.org/10.5555/555733
[22] Kumar, U., Kumar, V., & Kapur, J. N. (1986). Normalized measures f entropy. *International Journal of General Systems*, *12*(1), 55–69. https://doi.org/10.1080/03081078608934927
[23] Lang, K. (1998). Evidence driven state merging with search. *Rapport technique TR98–139, NECI*, *31*.
[24] Lassen, K. B., & van der Aalst, W. M. P. (2009). Complexity metrics for workflow nets. *Information and Software Technology*, *51*(3), 610–626. https://doi.org/10.1016/j.infsof.2008.08.005
[25] Leemans, S. J. J., Poppe, E., & Wynn, M. T. (2019). Directly follows-based process mining: Exploration & a case study. *2019 International Conference on Process Mining (ICPM)*, 25–32. https://doi.org/10.1109/ICPM.2019.00015
[26] Leemans, S. J., Fahland, D., & Aalst, W. M. (2018). Scalable process discovery and conformance checking. *Softw. Syst. Model.*, *17*(2), 599–631. https://doi.org/10.1007/s10270-016-0545-x
[27] Leemans, S. J. J. (2017). *Robust process mining with guarantees* (Thesis). https://doi.org/10.1007/978-3-030-96655-3
[28] Leemans, S. J. J., Fahland, D., & van der Aalst, W. M. P. (2013). Discovering block-structured process models from event logs - a constructive approach, 311–329. https://doi.org/10.1007/978-3-642-38697-8_17
[29] Leemans, S. J. J., Fahland, D., & van der Aalst, W. M. P. (2014). Discovering block-structured process models from event logs containing infrequent behaviour, 66–78. https://doi.org/10.1007/978-3-319-06257-0_6
[30] Lethbridge, T. C., Singer, J., & Forward, A. (2003). How software engineers use documentation: The state of the practice. *IEEE Software*, *20*(6), 35–39. https://doi.org/10.1109/MS.2003.1241364
[31] Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, *10*, 707–710.
[32] Lorenzoli, D., Mariani, L., & Pezzè, M. Automatic generation of software behavioral models. English. In: 2008, 501–510. ISBN: 9781605580791. https://doi.org/10.1145/1368088.1368157.
[33] Mariani, L., Pezzè, M., & Santoro, M. (2017). Gk-tail+ an efficient approach to learn software models. *IEEE Transactions on Software Engineering*, *43*(8), 715–738. https://doi.org/10.1109/TSE.2016.2623623
[34] McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, (4), 308–320. https://doi.org/10.1109/TSE.1976.233837
[35] Messaoudi, S., Panichella, A., Bianculli, D., Briand, L., & Sasnauskas, R. (2018). A search-based approach for accurate identification of log message formats. *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, 167–16710. https://doi.org/10.1145/3196321.3196340
[36] Mitchell, T. (1997). *Machine learning*. McGraw-Hill.
[37] Parnas, D. L. (1994). Software aging. *Proceedings of 16th International Conference on Software Engineering*, 279–287. https://doi.org/10.1109/ICSE.1994.296790
[38] Process discovery contest. (2021). https://doi.org/https://icpmconference.org/2021/process-discovery-contest/
[39] Process discovery contest: Example discovery algorithms. (2020). https://doi.org/https://icpmconference.org/2020/category/pdc-2021/example-discovery-algorithm/
[40] Quinlan, J. R. (2014). *C4.5: Programs for machine learning*. Elsevier. https://doi.org/https://doi.org/10.1007/BF00993309
[41] Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, *27*(3), 379–423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x
[42] Shin, D., Bianculli, D., & Briand, L. (2022). Prins: Scalable model inference for component-based system logs. *Empirical Software Engineering*, *27*(4), 87. https://doi.org/10.1007/s10664-021-10111-4
[43] Sipser, M. (2013). *Introduction to the theory of computation*. PWS Publishing Company. https://doi.org/10.5555/524279
[44] Soo, L. G., & Jung-Mo, Y. (1992). An empirical study on the complexity metrics of petri nets. *Microelectronics Reliability*, *32*(3), 323–329.

[45] Swart, A. A. H., Schult, D. A., & J., P. (2008). Exploring network structure, dynamics, and function using networkx. In G. V. Millman, T. Vaught, & Jarrod (Eds.), *Proceedings of the 7th python in science conference* (pp. 11–15). Los Alamos National Lab.(LANL), Los Alamos, NM (United States).
[46] Van Der Aalst, W. M. P. (1998). The application of petri nets to workflow management. *Journal of Circuits, Systems and Computers*, *08*(01), 21–66. https://doi.org/10.1142/S0218126698000043
[47] van der Aalst, W., van Dongen, B., Herbst, J., Maruster, L., Schimm, G., & Weijters, A. (2003). Workflow mining: A survey of issues and approaches. *Data and Knowledge Engineering*, *47*(2), 237–267. https://doi.org/10.1016/S0169-023X(03)00066-1
[48] Verbeek, H. M. W., Buijs, J. C. A. M., Van Dongen, B. F., & Van Der Aalst, W. M. P. (2010). Prom 6: The process mining toolkit. *615*, 34–39.
[49] Verbeek, H., Buijs, J. C., Van Dongen, B. F., & Van Der Aalst, W. M. (2010). Xes, xesame, and prom 6. *International Conference on Advanced Information Systems Engineering*, 60–75. https://doi.org/10.1007/978-3-642-17722-4_5
[50] Verwer, S., Eyraud, R., & de la Higuera, C. (2013). Pautomac: A probabilistic automata and hidden markov models learning competition. *Machine Learning*, *96*(1-2), 129–154. https://doi.org/10.1007/s10994-013-5409-9
[51] Verwer, S., & Hammerschmidt, C. (2022). Flexfringe: Modeling software behavior by learning probabilistic automata. *arXiv preprint arXiv:.16331*. https://doi.org/10.48550/arXiv.2203.16331
[52] Verwer, S., & Hammerschmidt, C. A. (2017). Flexfringe: A passive automaton learning package. https://doi.org/10.1109/icsme.2017.58
[53] Walkinshaw, N., Taylor, R., & Derrick, J. (2016). Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, *21*(3), 811–853. https://doi.org/10.1007/s10664-015-9367-7
[54] Zelst, v. S. J., Dongen, v. B. F., & Aalst, v. d. W. M. P. (2015a). Avoiding over-fitting in ilp-based process discovery, 163–171. https://doi.org/10.1007/978-3-319-23063-4_10
[55] Zelst, v. S. J., Dongen, v. B. F., & Aalst, v. d. W. M. P. (2015b). Ilp-based process discovery using hybrid regions.
[56] Zhu, J., He, S., Liu, J., He, P., Xie, Q., Zheng, Z., & Lyu, M. R. (2019). Tools and benchmarks for automated log parsing. https://doi.org/10.1109/icse-seip.2019.00021



Model inference tool settings

A.1 FlexFrings run settings

run command for Flexfringe

```
.\flexfringe.exe --ini ini/batch-aic.ini  path/to/input.csv --output_dir path/to/output
```

.ini file for Flexfringe

```
[default]
heuristic-name = aic
data-name = aic_data
state_count = 0
symbol_count = 0
satdfabound = 2000
largestblue = 1
printwhite = 1
printblue = 1
correction = 1
```

A.2 ProM run settings

Batch file to initiate prom with argument `-f path/to/script.txt`

```
@GOTO start

:add
 @set X=%X%;%1
 @GOTO :eof

:start
@set X=.\dist\ProM-Framework-6.11.121.jar
@set X=%X%;.\dist\ProM-Contexts-6.11.67.jar
@set X=%X%;.\dist\ProM-Models-6.10.43.jar
@set X=%X%;.\dist\ProM-Plugins-6.9.75.jar

@for /R .\lib %%I IN ("*.jar") DO @call :add .\lib\%%~nI.jar

@jre8\bin\java^
 -Xmx10G^
 -da^
 -classpath "%X%"^
 -Djava.library.path=.//lib^
 -Djava.system.class.loader=org.processmining.framework.util.ProMClassLoader^
 -Djava.util.Arrays.useLegacyMergeSort=true^
 org.processmining.contexts.cli.CLI %1 %2

set X=
```

Script file for running ProM

```
// Code examples by:
//          https://github.com/DStekel3/ProM-CommandLine-Scripts
//          https://dirksmetric.wordpress.com/2015/03/11/tutorial-automating-process-mining-
    with-proms-command-line-interface/
//          https://www.win.tue.nl/~hverbeek/pdc2020/miners/
import java.io.File;
import org.apache.commons.lang3.ArrayUtils;

import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;
import org.apache.commons.io.output.NullOutputStream;

import java.util.Arrays;

import org.apache.commons.io.FileUtils;


xesPath = "C:\\path\\to\\input\\folder\\";
outputPath = "C:\\path\\to\\output\\folder\\";

xesFile = "C:\\path\\to\\xes\\file.xes";

String fileName = xesFile.getName().split""(\\.);[0]

FileWriter writer = new FileWriter("path/to/runtime/file.txt");
BufferedWriter bw = new BufferedWriter(writer);

// Read XES file
org.deckfour.xes.model.XLog log = open_xes_log_file(xesFile);

// INDUCTIVE
print("Start inductive miner");
long start =  System.nanoTime();
// Calling parameters for miners
org.processmining.plugins.InductiveMiner.mining.MiningParametersIM parameters = new org.
    processmining.plugins.InductiveMiner.mining.MiningParametersIM();

ind_net = mine_petri_net_with_inductive_miner_with_parameters(log, parameters);

long end = System.nanoTime();
long t = end - start;
bw.write(fileName + " inductive " + t);
bw.newLine();

// Saving net
File net_file = new File(outputPath+ fileName + "_inductive.pnml");
pnml_export_petri_net_(ind_net[0], net_file);

bw.close();
// Close ProM CLI
exit();
```

B Pseudo code

B.1 Perplexity for Petri nets

Perplexity for Petri nets

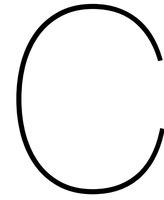
```
// net: is a PM4Py Petri net object
// im, fm: the initial and final marking of the net
// test_data: set of test traces
// Returns perplexity on the test data
// TBR output of PM4Py was modified to return:
// A list of travelled markings, a marking is a list of places in that marking.

import modified_token_replay as token_replay

PERPLEXITY_PETRI_NET(net, im, fm, test_data):
    alphabet_size = count_unique_transitions(net)        // Get alphabet size of net
    normalisation = 1 / (1 + (1 / alphabet_size))        // Normalisation factor
    unseen = (1 / alphabet_size) * normalisation         // Probability of unseen event

    symbols_seen = 0
    result = 0

    // Get our perplexity per trace, add each to result
    for t_log in test_data:

        followed_markings, activated_transitions, is_fit = token_replay.apply(t_log, net, im,
             fm, parameters={ 'stop_immediately_unfit':1})

        prob_log = 0

        for marking in followed_markings:
            // If there is no followed path left, and the trace is not fit,
            // We are stuck, penalise with unseen and break.
            // If the trace is fit, no penalty is needed and we can break
            if length(followed_path) == 0:
                if not is_fit:
                    prob_log = prob_log + math.log2(unseen)
                break

            possible_transitions = []
            current_transition = followed_path.pop(0)

            for place in marking:
                // Loop through outgoing arcs of a place and add their target transitions
                for out_arc in place.out_arcs:
                    possible_transitions.append(out_arc.target)

            // If the transition is in the possible transitions,
            // We move forward with it and add its probability
            if current_transition in possible_transitions:
                // Count number of times the transition occurs as out transition
```

```
46                  curr_trans_occurrence = possible_transitions.count(current_transition)
47                  total_out_edges = length(possible_transitions)
48                  // If the transition is a tau (None), its occurrence is 1
49                  if current_transition is None:
50                      curr_trans_occurrence = 1
51
52                  symbols_seen = symbols_seen + 1
53                  probability = (curr_trans_occurrences / total_out_edges ) * normalisation
54                  prob_log = prob_log + math.log2(probability)
55
56              // If we got stuck, add unseen probability and break
57              if current_transition not in possible_transitions:
58                  prob_log = prob_log + math.log2(unseen)
59                  break
60
61          result = result + prob_log
62
63      result = result / symbols_seen
64
65      return math.pow(2, -result)
```

B.2 Token-based replay for graphs

Token-based fitness for graphs

```
1   // Computes TBR fitness with number of produced, consumed and missing tokens.
2   GET_TBR_FITNESS(p, c, m):
3       r = p + m - c
4       return 0.5 * (1 - (r / p)) + 0.5 * (1 - (m / c))
5
6   // A path is a sequence of transition labels
7   // Returns TRUE if path complies to model, False otherwise
8   // Returns token based fitness
9   TOKEN_REPLAY_GRAPH(graph, path):
10      p = 0
11      c = 0
12      m = 0
13      queue = [graph.root]
14      while len(path) > 0:
15          // If we begin a loop, we must be in a state. This means we traversed a transition.
16          // Therefore, a token was consumed by the previous traveled transition,
17          // and the transition always has 1 edge to the state, so it produced 1 token.
18          p = p + 1
19          c = c + 1
20          transition = path.pop(0)
21          new_queue = []
22
23          while len(queue) > 0:
24              node = queue.pop(0)
25              for each outgoing_transition of node:
26                  if outgoing_transition == transition
27                      add to new_queue
28
29          if len(new_queue) == 0:              // Path is stuck
30              m = m + 1
31              return False, GET_TBR_FITNESS()
32
33
34          queue = new_queue.copy()
35
36          if len(path) == 0 and len(queue) != 0:        // End of path was reached
37              if any node in queue is accept_states:
38                  return True, 1
39              else
40                  m = m + 1
41                  return False, GET_TBR_FITNESS
```



Full results

This appendix contains one table with the recall of the FSM on their training data and 2 large tables containing all other results.

Log name	FF	PRINS		MINT
	AIC	W4-HD1	W4-NFA	ADB-2
CoreSync	1.0	0.98	0.98	0.99
Hadoop	1.0	1.00	1.00	1.00
HDFS	1.0	1.00	0.99	1.00
Linux	1.0	0.76	0.79	0.87
NGLClient	1.0	0.98	0.97	0.97
Oobelib	1.0	-	-	-
PDApp	1.0	0.99	0.99	0.99
Spark	1.0	1.00	1.00	1.00
Zookeeper	1.0	0.77	0.79	0.85

Table C.1: Recall of FSM models on their *training data set*, to detect overfitting

Log name	Program	Run config	Run time	Sound	States	Transitions	BA	f_2-score	Recall	Precision	Accuracy	Specificity	TBR fitness	Perplexity	Align fitness	Align cost	$eCFC$	AVG $\%trav$	CC	#Traces	#Events	$\lvert\overline{T}\rvert$	$\overline{S}$	$H_\eta(E)$
CoreSync	FF	AIC	17.853	1.0	500.8	822.4	0.918	0.925	0.929	0.909	0.918	0.907	0.986	1.996	0.993	1869.46	1.641	0.958	323.6	283.0	153.2	21.34	0.19	0.478
	MINT	ADB-2	4106.746	0.0	3957.6	4794.0	0.862	0.939	0.986	0.79	0.862	0.738	0.997	2.636	0.999	108.383	1.211	0.991	838.4	283.0	153.2	21.34	0.19	0.478
	PRINS	W4-HD1	2062.462	0.0	7194.0	10853.4	0.894	0.947	0.979	0.836	0.894	0.809	0.997	2.479	-	-	1.493	0.989	3661.4	283.0	153.2	21.34	0.19	0.478
		W4-NFA	254.023	-42.0	17341.8	20139.2	0.903	0.952	0.983	0.848	0.903	0.823	0.997	1.93	-	-	1.162	0.988	2799.4	283.0	153.2	21.34	0.19	0.478
	ProM	DF	0.128	1.0	51.6	70.6	0.81	0.833	0.845	0.79	0.81	0.775	0.885	1.819	0.871	144149.41	1.155	0.847	21.0	283.0	153.2	21.34	0.19	0.478
		Hybrid-ILP	2.709	0.4	8.6	8.2	0.64	0.456	0.421	0.749	0.64	0.86	0.465	2.527	0.441	213005.534	1.173	0.428	4.0	283.0	153.2	21.34	0.19	0.478
		IM	2.576	1.0	547.6	700.2	0.506	0.252	0.236	0.53	0.506	0.777	0.938	109.809	0.999	235.549	1.234	0.754	402.2	283.0	153.2	21.34	0.19	0.478
		IMfa	1.784	1.0	419.2	527.8	0.583	0.596	0.613	0.571	0.583	0.553	0.981	70.345	0.988	1756.336	1.4	0.804	347.0	283.0	153.2	21.34	0.19	0.478
HDFS	FF	AIC	0.509	1.0	86.4	164.2	0.939	0.951	0.959	0.922	0.939	0.919	0.995	1.935	0.996	832.0	1.895	0.98	79.8	200.0	14.4	18.72	0.662	0.249
	MINT	ADB-2	210.368	1.0	55.8	186.2	0.682	0.883	0.993	0.613	0.682	0.37	0.999	8.028	0.999	82.0	3.34	0.998	132.4	200.0	14.4	18.72	0.662	0.249
	PRINS	W4-HD1	8.847	0.8	537.6	4130.8	0.79	0.919	0.994	0.71	0.79	0.585	0.999	6.445	1.0	77.0	6.121	0.997	3595.2	200.0	14.4	18.72	0.662	0.249
		W4-NFA	6.544	0.0	5839.8	9381.8	0.817	0.926	0.99	0.742	0.816	0.643	0.999	4.485	-	-	1.688	0.996	3544.0	200.0	14.4	18.72	0.662	0.249
	ProM	DF	0.114	1.0	10.2	24.4	0.81	0.875	0.911	0.758	0.81	0.709	0.983	2.882	0.98	3841.0	1.41	0.919	16.2	200.0	14.4	18.72	0.662	0.249
		Hybrid-ILP	0.417	0.0	8.0	9.0	0.616	0.279	0.237	0.979	0.616	0.995	0.876	3.761	0.843	29932.0	2.824	0.37	9.0	200.0	14.4	18.72	0.662	0.249
		IM	0.398	1.0	51.0	64.2	0.631	0.791	0.882	0.582	0.631	0.381	0.999	10.478	1.0	143.742	1.23	0.962	34.4	200.0	14.4	18.72	0.662	0.249
		IMfa	0.144	1.0	22.0	29.2	0.735	0.869	0.941	0.667	0.735	0.529	0.993	3.008	0.953	8950.186	1.325	0.897	18.4	200.0	14.4	18.72	0.662	0.249
Hadoop	FF	AIC	0.084	1.0	53.0	54.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.06	1.0	2.0	1.019	1.0	3.0	13.0	41.0	52.6	0.984	0.531
	MINT	ADB-2	1.813	1.0	139.0	144.8	0.962	0.985	1.0	0.929	0.962	0.923	1.0	1.111	1.0	2.0	1.041	1.0	7.8	13.0	41.0	52.6	0.984	0.531
	PRINS	W4-HD1	4.023	1.0	65.0	68.0	0.923	0.97	1.0	0.867	0.923	0.846	1.0	1.369	1.0	2.0	1.045	1.0	5.0	13.0	41.0	52.6	0.984	0.531
		W4-NFA	3.745	0.0	2149.0	2315.2	0.923	0.97	1.0	0.867	0.923	0.846	1.0	1.335	1.0	2.0	1.077	1.0	168.2	13.0	41.0	52.6	0.984	0.531
	ProM	DF	0.08	1.0	43.0	49.2	0.877	0.872	0.877	0.88	0.877	0.877	0.99	1.438	0.998	1847.154	1.067	0.918	8.2	13.0	41.0	52.6	0.984	0.531
		Hybrid-ILP	0.462	0.0	41.0	42.0	0.823	0.806	0.8	0.837	0.823	0.846	0.984	1.697	0.993	5848.154	1.301	0.747	17.0	13.0	41.0	52.6	0.984	0.531
		IM	0.245	1.0	43.0	47.0	0.962	0.985	1.0	0.929	0.962	0.923	1.0	1.396	1.0	12.938	1.1	1.0	8.0	13.0	41.0	52.6	0.984	0.531
		IMfa	0.043	1.0	43.0	47.0	0.962	0.985	1.0	0.929	0.962	0.923	1.0	1.396	1.0	12.938	1.1	1.0	8.0	13.0	41.0	52.6	0.984	0.531
Linux	FF	AIC	15.056	1.0	348.0	696.4	0.35	0.0	0.0	0.0	0.35	0.7	0.92	2.838	0.961	50002.0	2.0	0.24	350.4	8.0	108.2	273.56	0.238	0.277
	MINT	ADB-2	235.033	1.0	272.8	419.6	0.675	0.778	0.825	0.635	0.675	0.525	0.991	4.326	0.999	4323.428	1.536	0.881	148.8	8.0	108.2	273.56	0.238	0.277
	PRINS	W4-HD1	163.188	0.0	1742.8	5147.2	0.812	0.818	0.825	0.805	0.812	0.8	0.994	3.79	-	-	2.651	0.881	3406.4	8.0	108.2	273.56	0.238	0.277
		W4-NFA	78.765	0.0	1535.4	2401.0	0.763	0.749	0.75	0.768	0.763	0.775	0.984	3.445	-	-	1.572	0.812	867.6	8.0	108.2	273.56	0.238	0.277
	ProM	DF	0.092	1.0	115.4	298.2	0.475	0.402	0.4	0.437	0.475	0.55	0.92	7.552	0.976	36893.857	1.442	0.549	184.8	8.0	108.2	273.56	0.238	0.277
		Hybrid-ILP	9.025	0.0	37.2	22.0	0.6	0.476	0.45	0.629	0.6	0.75	0.665	23.794	0.873	235502.0	249.054	0.505	417.4	8.0	108.2	273.56	0.238	0.277
		IM	0.49	1.0	219.4	215.2	0.45	0.483	0.525	0.379	0.45	0.375	0.976	31.447	0.993	9132.64	1.133	0.79	110.2	8.0	108.2	273.56	0.238	0.277
		IMfa	0.241	1.0	181.8	163.2	0.5	0.084	0.075	0.28	0.5	0.925	0.926	19.212	0.942	52851.625	1.124	0.313	85.8	8.0	108.2	273.56	0.238	0.277

Table C.2: All results, 5-folds aggregated: part 1.
#Traces and all columns after were calculated on test folds.

Log name	Program	Run config	Run time	Sound	States	Transitions	BA	f_2-score	Recall	Precision	Accuracy	Specificity	TBR fitness	Perplexity	Align fitness	Align cost	$eCFC$	AVG $\%trav$	CC	#Traces	#Events	$\lvert\overline{T}\rvert$	$\overline{S}$	$H_\eta(E)$
NGLClient	FF	`AIC`	0.116	1.0	91.0	106.0	0.938	0.895	0.875	1.0	0.938	1.0	0.952	1.318	0.937	45752.0	1.161	0.926	17.0	8.0	44.8	21.28	0.272	0.666
	MINT	`ADB-2`	1.62	1.0	136.6	146.6	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.493	1.0	2.0	1.073	1.0	12.0	8.0	44.8	21.28	0.272	0.666
	PRINS	`W4-HD1`	4.538	1.0	140.6	175.6	0.862	0.918	0.95	0.809	0.862	0.775	0.993	1.681	0.977	4002.0	1.243	0.98	37.0	8.0	44.8	21.28	0.272	0.666
		`W4-NFA`	4.169	1.0	508.6	650.6	0.888	0.957	1.0	0.818	0.888	0.775	1.0	1.589	1.0	2.0	1.278	1.0	144.0	8.0	44.8	21.28	0.272	0.666
	ProM	`DF`	0.076	1.0	48.8	64.4	0.825	0.804	0.8	0.836	0.825	0.85	0.844	1.761	0.889	44251.0	1.138	0.852	17.6	8.0	44.8	21.28	0.272	0.666
		`Hybrid-ILP`	0.619	0.8	18.8	18.0	0.675	0.399	0.35	1.0	0.675	1.0	0.542	1.917	0.484	163502.0	1.233	0.407	12.6	8.0	44.8	21.28	0.272	0.666
		`IM`	0.336	1.0	135.8	162.2	0.7	0.816	0.875	0.647	0.7	0.525	0.988	38.75	0.98	9551.921	1.21	0.935	100.4	8.0	44.8	21.28	0.272	0.666
		`IMfa`	0.082	1.0	97.6	117.0	0.675	0.741	0.775	0.647	0.675	0.575	0.962	23.612	0.971	5249.393	1.205	0.871	71.0	8.0	44.8	21.28	0.272	0.666
Oobelib	FF	`AIC`	19.195	1.0	616.2	822.8	0.884	0.819	0.788	0.975	0.884	0.98	0.99	1.455	0.997	3602.0	1.334	0.863	208.6	50.0	117.4	226.2	0.612	0.34
	MINT	`ADB-2`	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	50.0	117.4	226.2	0.612	0.34
	PRINS	`W4-HD1`	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	50.0	117.4	226.2	0.612	0.34
		`W4-NFA`	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	50.0	117.4	226.2	0.612	0.34
	ProM	`DF`	1.022	1.0	91.0	199.4	0.758	0.732	0.724	0.775	0.758	0.792	0.935	4.899	0.984	32321.0	1.373	0.809	110.4	50.0	117.4	226.2	0.612	0.34
		`Hybrid-ILP`	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	50.0	117.4	226.2	0.612	0.34
		`IM`	8.997	1.0	380.8	449.8	0.688	0.877	0.98	0.618	0.688	0.396	0.998	169.539	-	-	1.229	0.982	305.8	50.0	117.4	226.2	0.612	0.34
		`IMfa`	6.135	1.0	270.6	320.2	0.558	0.347	0.324	0.487	0.558	0.792	0.781	98.992	-	-	1.26	0.254	205.2	50.0	117.4	226.2	0.612	0.34
PDApp	FF	`AIC`	3.037	1.0	404.8	472.2	0.984	0.974	0.968	0.999	0.983	0.999	0.996	1.186	0.994	2691.94	1.166	0.984	69.4	157.0	60.0	60.24	0.43	0.334
	MINT	`ADB-2`	683.899	0.2	1068.6	1241.6	0.968	0.98	0.987	0.951	0.968	0.949	0.999	2.221	0.999	481.238	1.187	0.993	175.0	157.0	60.0	60.24	0.43	0.334
	PRINS	`W4-HD1`	236.718	0.0	3048.25	3423.0	0.979	0.985	0.989	0.97	0.98	0.97	1.0	1.538	-	-	1.122	0.996	376.75	157.0	60.0	60.24	0.43	0.334
		`W4-NFA`	1196.317	-42.0	29374.6	33737.2	0.968	0.974	0.978	0.959	0.968	0.958	0.998	1.257	-	-	1.15	0.99	4364.6	157.0	60.0	60.24	0.43	0.334
	ProM	`DF`	0.133	1.0	40.0	68.0	0.865	0.848	0.839	0.886	0.866	0.892	0.948	2.88	0.956	64739.854	1.259	0.873	30.0	157.0	60.0	60.24	0.43	0.334
		`Hybrid-ILP`	1.059	0.0	19.0	19.0	0.727	0.584	0.541	0.858	0.726	0.911	0.479	7.203	0.685	305973.337	12.395	0.573	68.6	157.0	60.0	60.24	0.43	0.334
		`IM`	1.756	1.0	228.8	291.4	0.505	0.725	0.843	0.494	0.505	0.166	0.982	80.993	-	-	1.239	0.991	173.0	157.0	60.0	60.24	0.43	0.334
		`IMfa`	0.645	1.0	114.6	143.2	0.538	0.516	0.525	0.544	0.538	0.552	0.957	41.641	-	-	1.225	0.28	86.6	157.0	60.0	60.24	0.43	0.334
Spark	FF	`AIC`	3.913	1.0	39.8	53.2	0.877	0.939	0.977	0.816	0.877	0.777	1.0	1.687	1.0	643.026	1.258	0.992	15.4	43.0	17.4	309.96	0.43	0.038
	MINT	`ADB-2`	2580.216	1.0	21.667	30.333	0.849	0.943	1.0	0.768	0.849	0.698	1.0	1.187	1.0	2.0	1.38	1.0	10.667	43.0	17.4	309.96	0.43	0.038
	PRINS	`W4-HD1`	1560.157	1.0	20.0	25.0	0.849	0.943	1.0	0.768	0.849	0.698	1.0	2.972	1.0	2.0	1.238	1.0	7.0	43.0	17.4	309.96	0.43	0.038
		`W4-NFA`	872.06	0.0	1366.0	1826.5	0.849	0.943	1.0	0.768	0.849	0.698	1.0	1.987	-	-	1.337	1.0	462.5	43.0	17.4	309.96	0.43	0.038
	ProM	`DF`	0.086	1.0	17.0	20.0	0.856	0.94	0.991	0.78	0.856	0.721	0.999	3.022	0.999	4466.116	1.081	0.991	5.0	43.0	17.4	309.96	0.43	0.038
		`Hybrid-ILP`	2.23	1.0	15.0	14.8	0.805	0.841	0.875	0.757	0.805	0.735	0.993	7.463	0.894	827397.349	2.314	0.881	18.6	43.0	17.4	309.96	0.43	0.038
		`IM`	0.532	1.0	24.0	30.0	0.83	0.936	1.0	0.746	0.83	0.66	1.0	2.301	1.0	294.865	1.13	1.0	10.0	43.0	17.4	309.96	0.43	0.038
		`IMfa`	0.283	1.0	24.0	27.0	0.765	0.604	0.554	0.959	0.765	0.977	0.978	2.608	0.974	73922.316	1.078	0.59	7.0	43.0	17.4	309.96	0.43	0.038
Zookeeper	FF	`AIC`	32.754	1.0	280.8	685.2	0.443	0.162	0.143	0.433	0.443	0.743	0.966	3.301	0.952	37502.0	2.424	0.399	406.4	7.0	36.8	715.18	0.194	0.235
	MINT	`ADB-2`	2863.829	0.333	344.75	574.0	0.607	0.733	0.786	0.581	0.607	0.429	0.992	5.794	0.996	5002.0	1.676	0.85	231.25	7.0	36.8	715.18	0.194	0.235
	PRINS	`W4-HD1`	78.362	0.2	1785.4	4741.2	0.742	0.794	0.828	0.738	0.743	0.657	0.994	4.734	-	-	2.61	0.905	2957.8	7.0	36.8	715.18	0.194	0.235
		`W4-NFA`	69.275	0.0	2087.2	4381.4	0.714	0.754	0.771	0.698	0.714	0.657	0.99	4.318	-	-	2.113	0.824	2296.2	7.0	36.8	715.18	0.194	0.235
	ProM	`DF`	0.338	1.0	42.0	172.0	0.486	0.462	0.457	0.492	0.486	0.514	0.981	6.021	0.994	5001.0	1.607	0.656	132.0	7.0	36.8	715.18	0.194	0.235
		`Hybrid-ILP`	247.842	0.0	44.0	36.0	0.5	0.167	0.143	0.5	0.5	0.857	0.9	3.972	-	-	93.862	0.182	271.0	7.0	36.8	715.18	0.194	0.235
		`IM`	2.095	1.0	126.0	157.2	0.5	0.674	0.743	0.497	0.5	0.257	0.995	49.141	1.0	152.775	1.21	0.893	88.0	7.0	36.8	715.18	0.194	0.235
		`IMfa`	1.435	1.0	108.8	136.8	0.514	0.388	0.372	0.514	0.514	0.657	0.948	34.575	0.995	2613.25	1.357	0.6	88.8	7.0	36.8	715.18	0.194	0.235

Table C.3: All results, 5-folds aggregated: part 2.
#Traces and all columns after were calculated on test folds.

D

Data Composition and Performance

This appendix contains graphs with BA, F_2-score, fitness (f_{AL}, f_{TBR}), perplexity and cyclomatic complexity CC were plotted against trace similarity, normalised entropy and number of unique events. These graphs are done per run configuration.

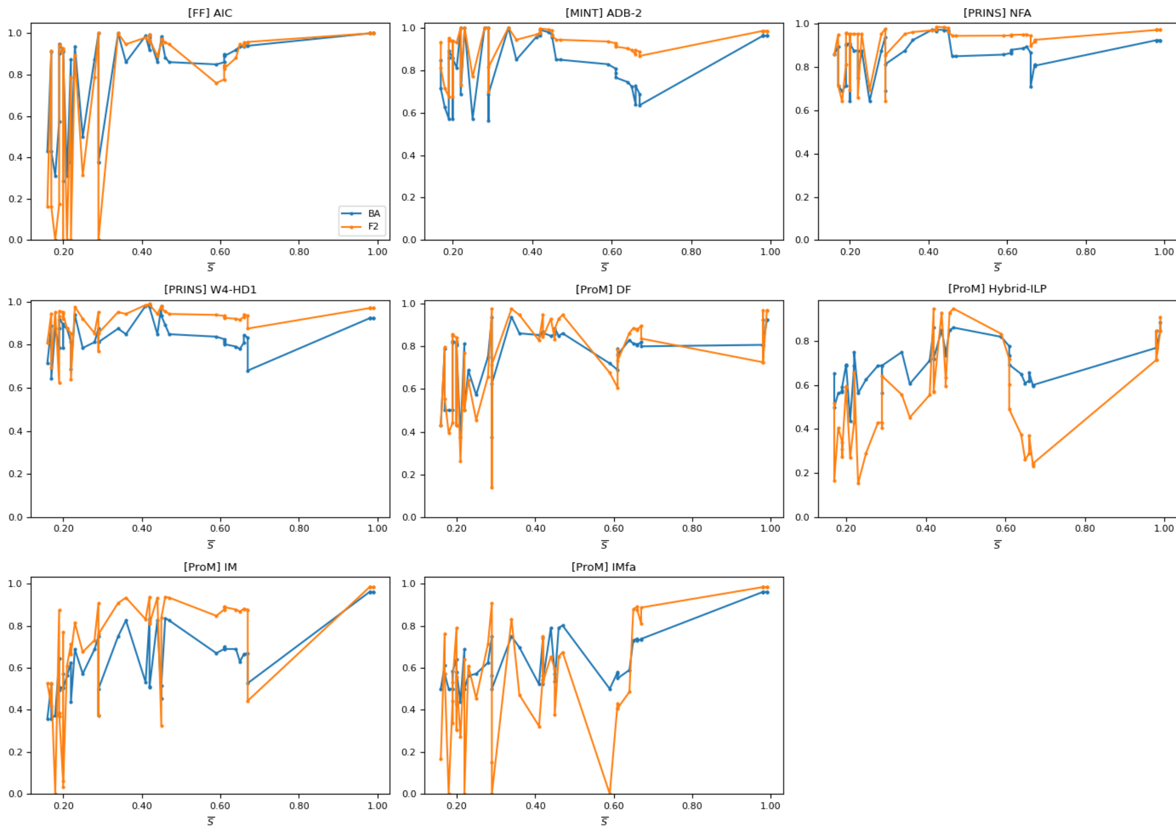


Figure D.1: BA, F_2-score vs trace similarity $\overline{S}$

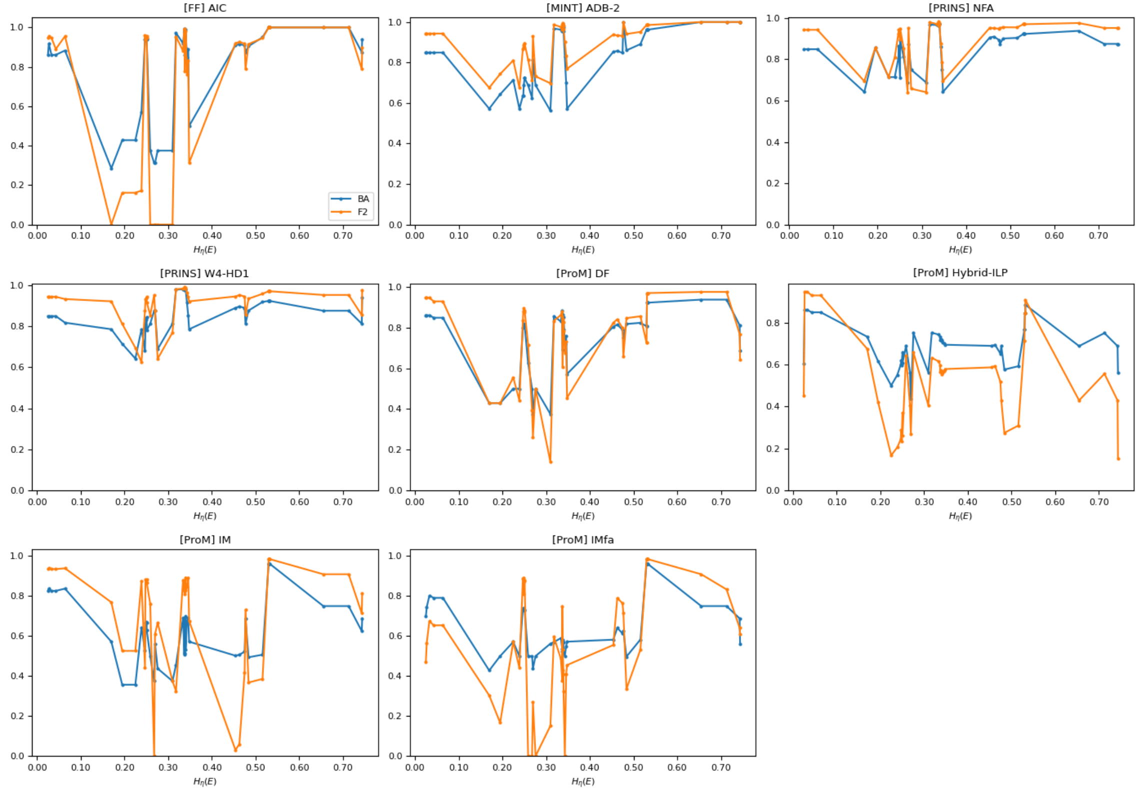


Figure D.2: BA, F_2-score vs normalised entropy $H_\eta(e)$

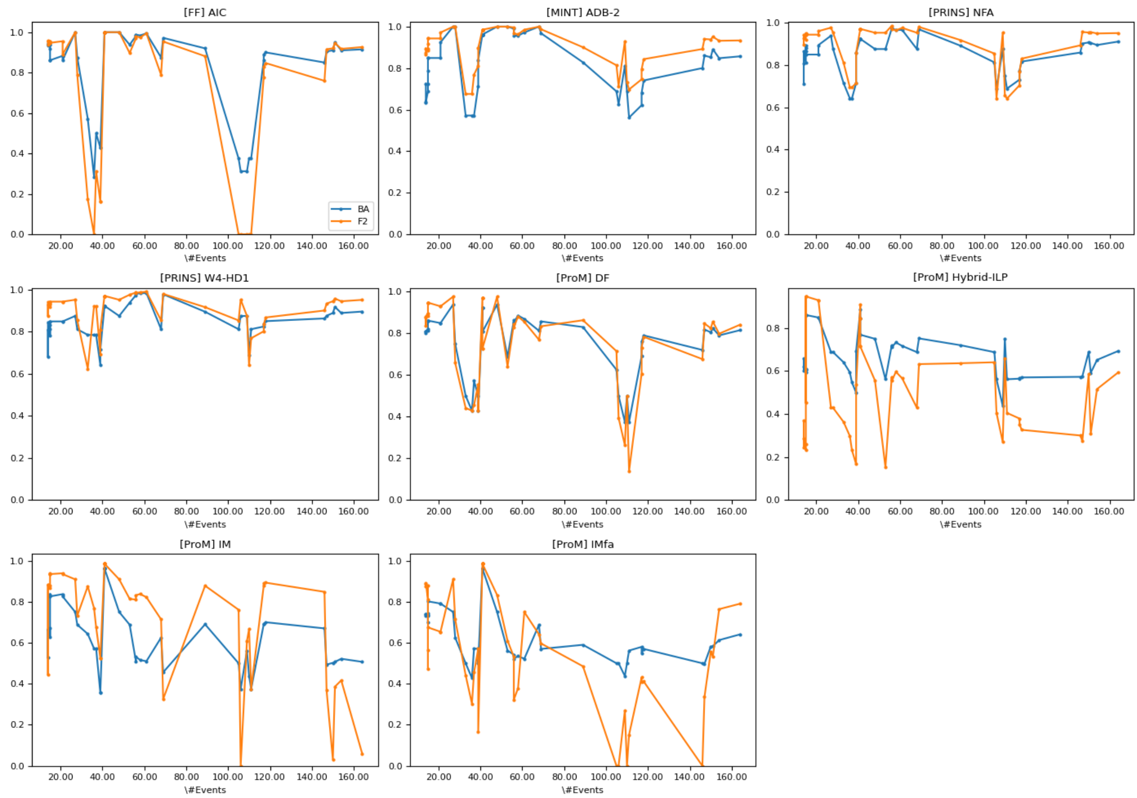


Figure D.3: BA, F_2-score vs unique events $\#Events$

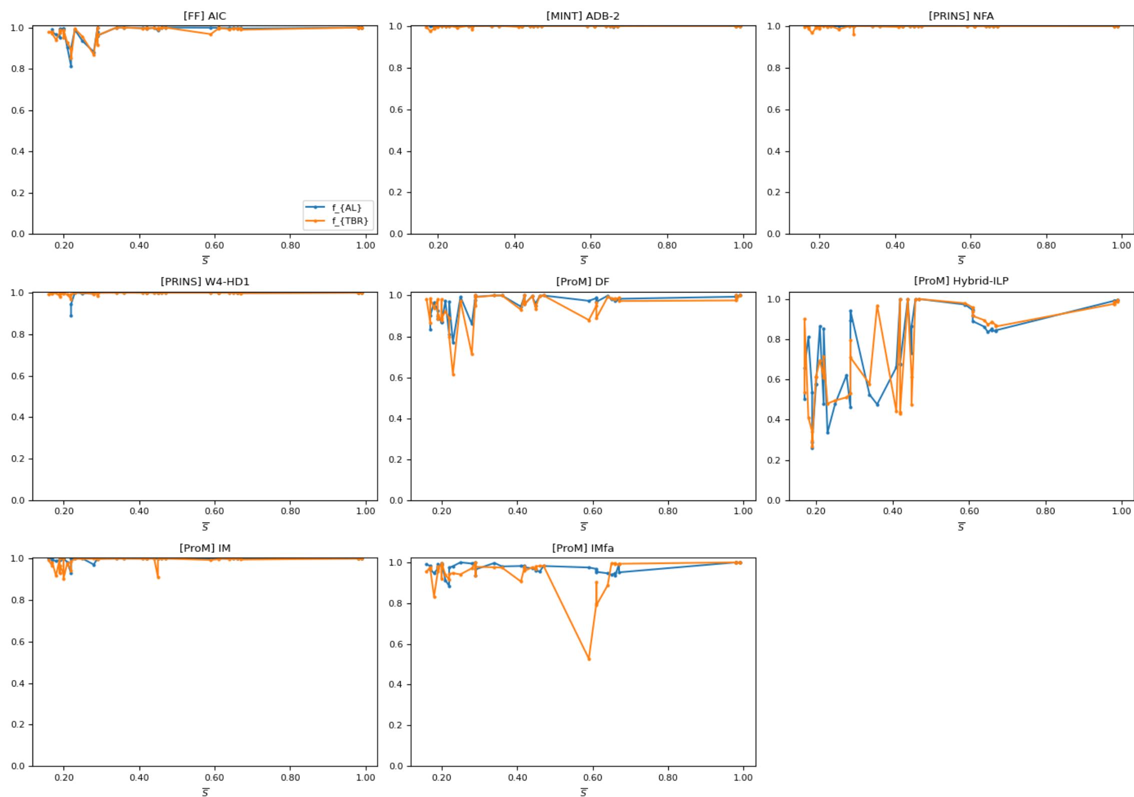


Figure D.4: Align and TBR fitness f_{AL}, f_{TBR} vs trace similarity $\overline{S}$

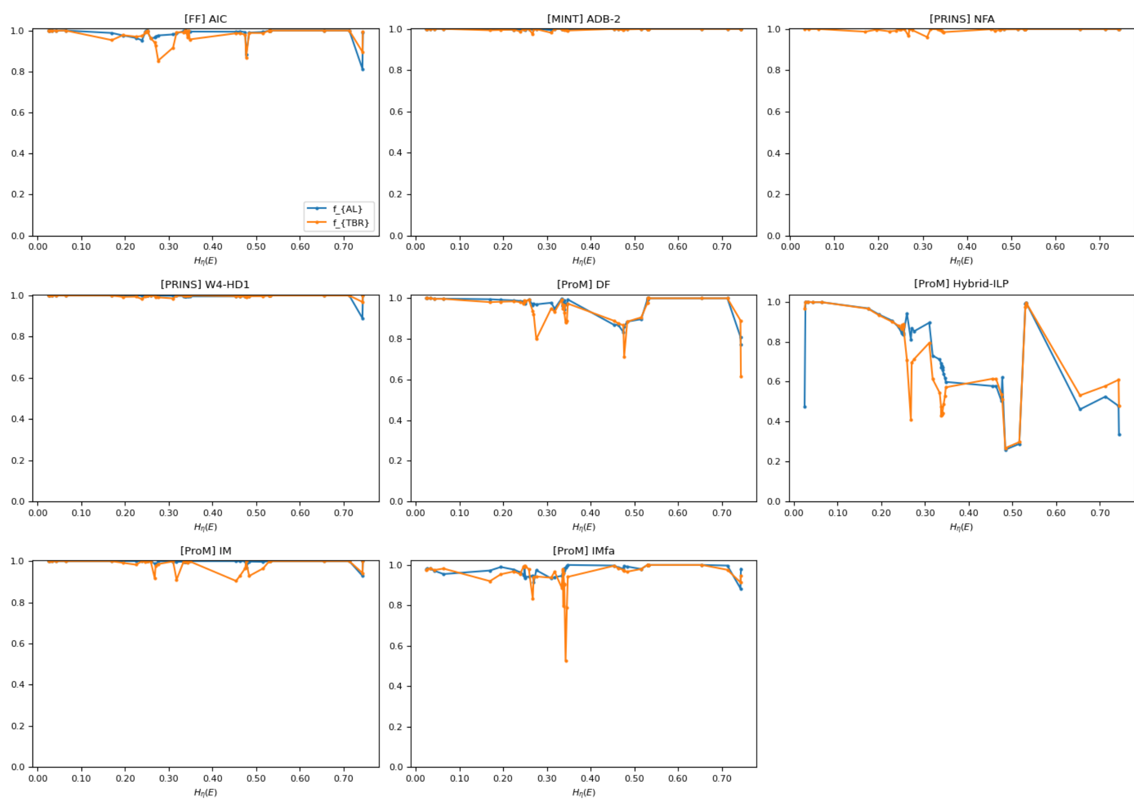


Figure D.5: Align and TBR fitness f_{AL}, f_{TBR} vs normalised entropy $H_{\eta}(e)$

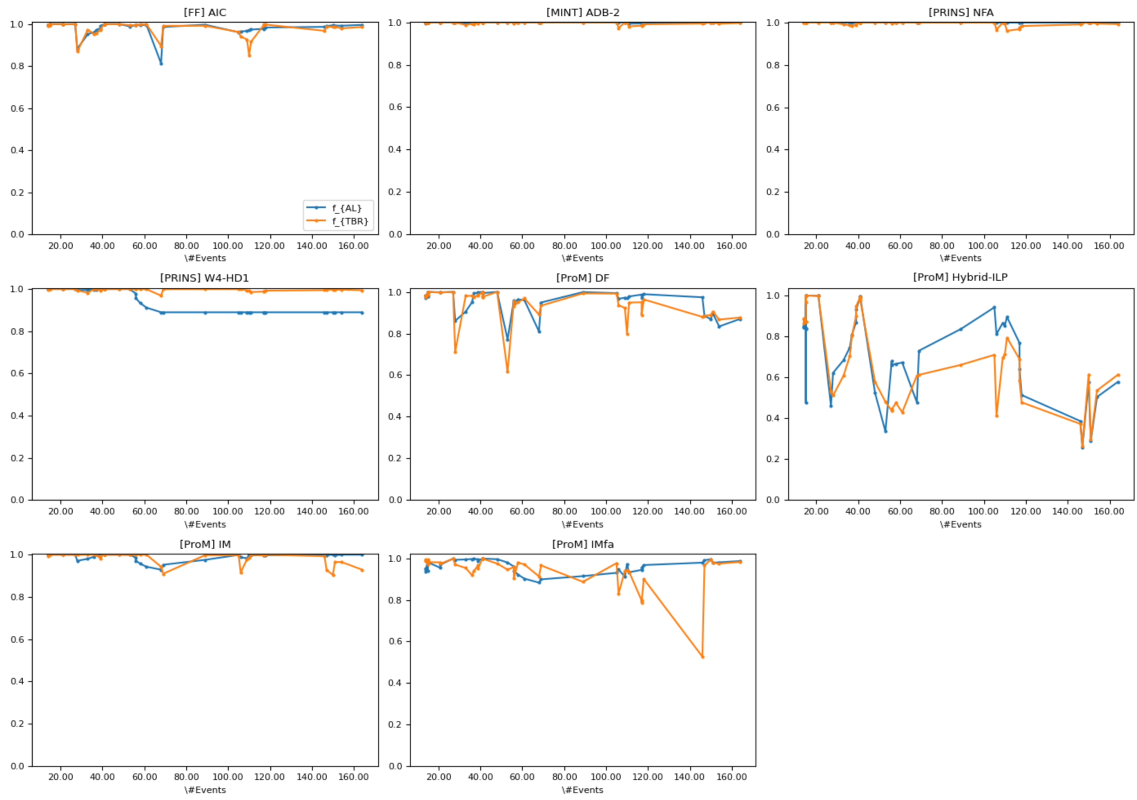


Figure D.6: Align and TBR fitness f_{AL}, f_{TBR} vs unique events $\#Events$

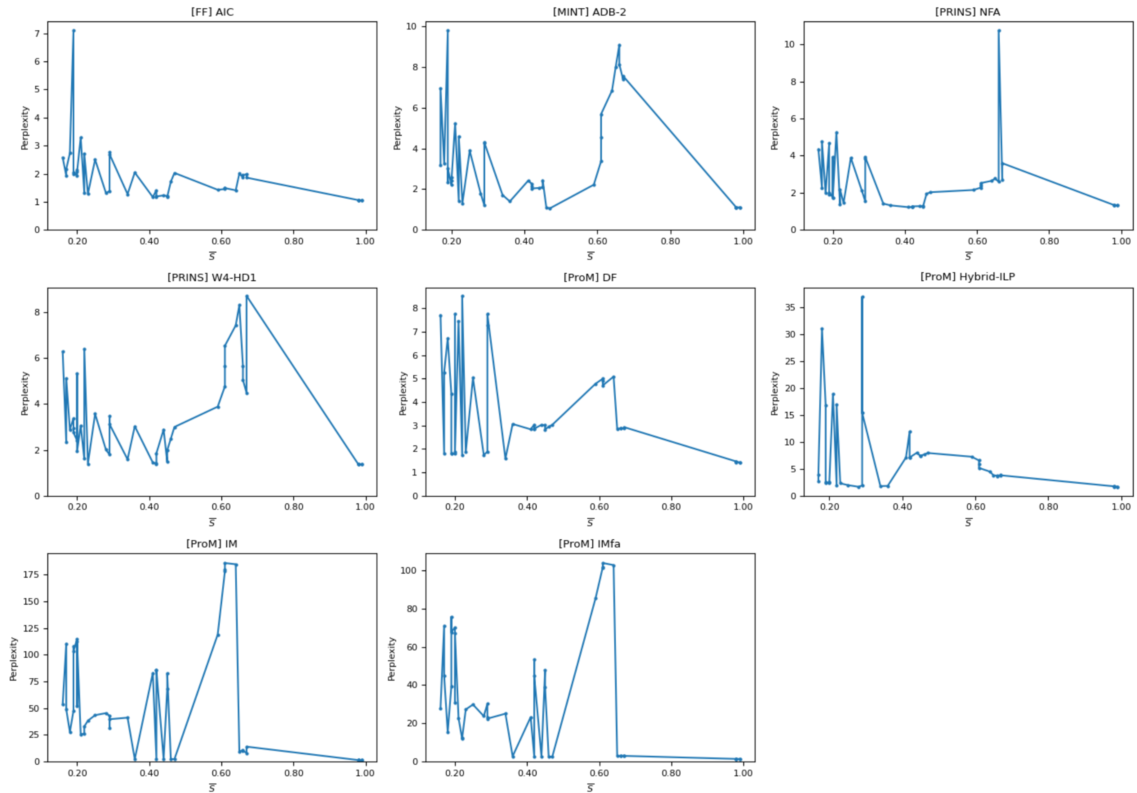


Figure D.7: Perplexity vs trace similarity $\overline{S}$

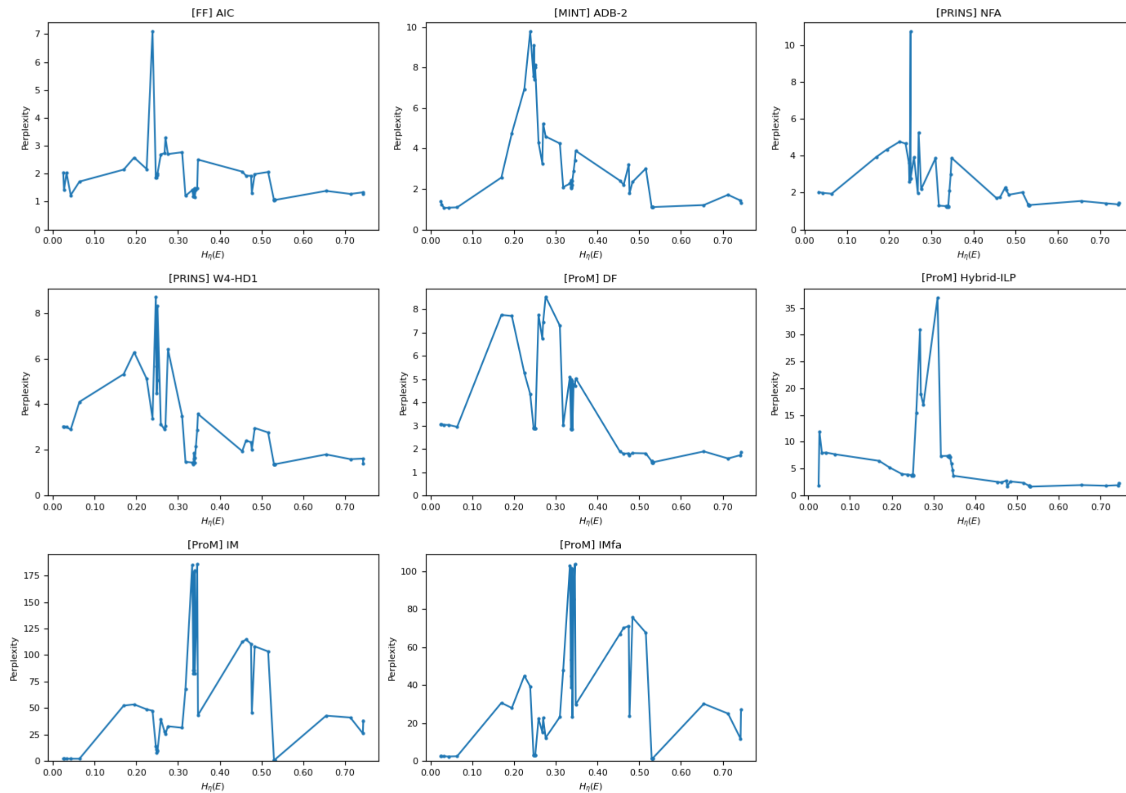


Figure D.8: Perplexity vs normalised entropy $H_\eta(e)$

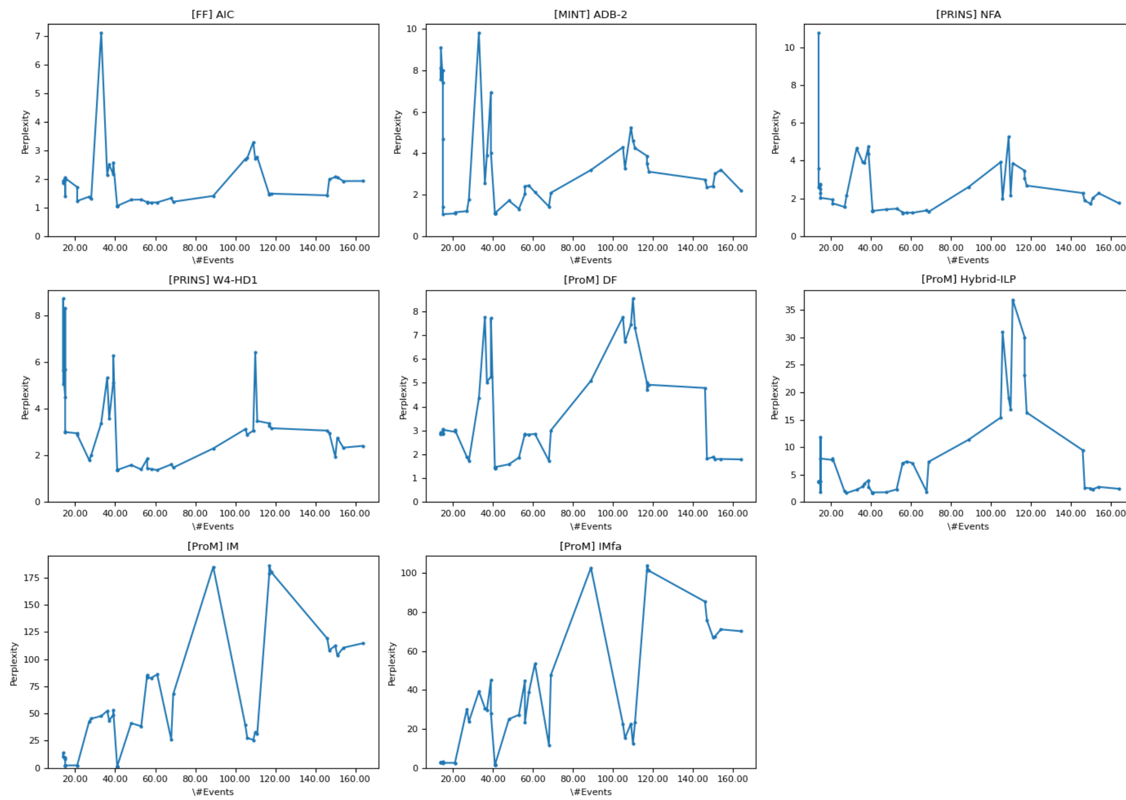


Figure D.9: Perplexity vs unique events $\#Events$

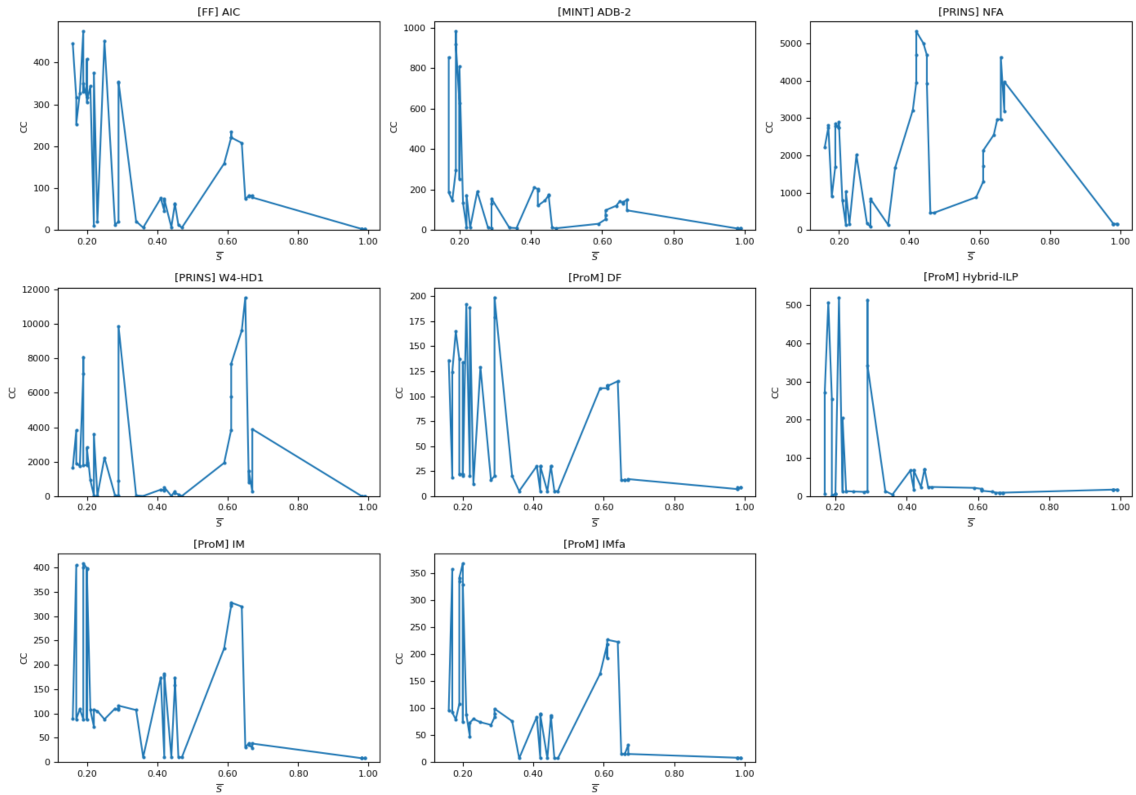


Figure D.10: Cyclomatic Complexity CC vs trace similarity $\overline{S}$

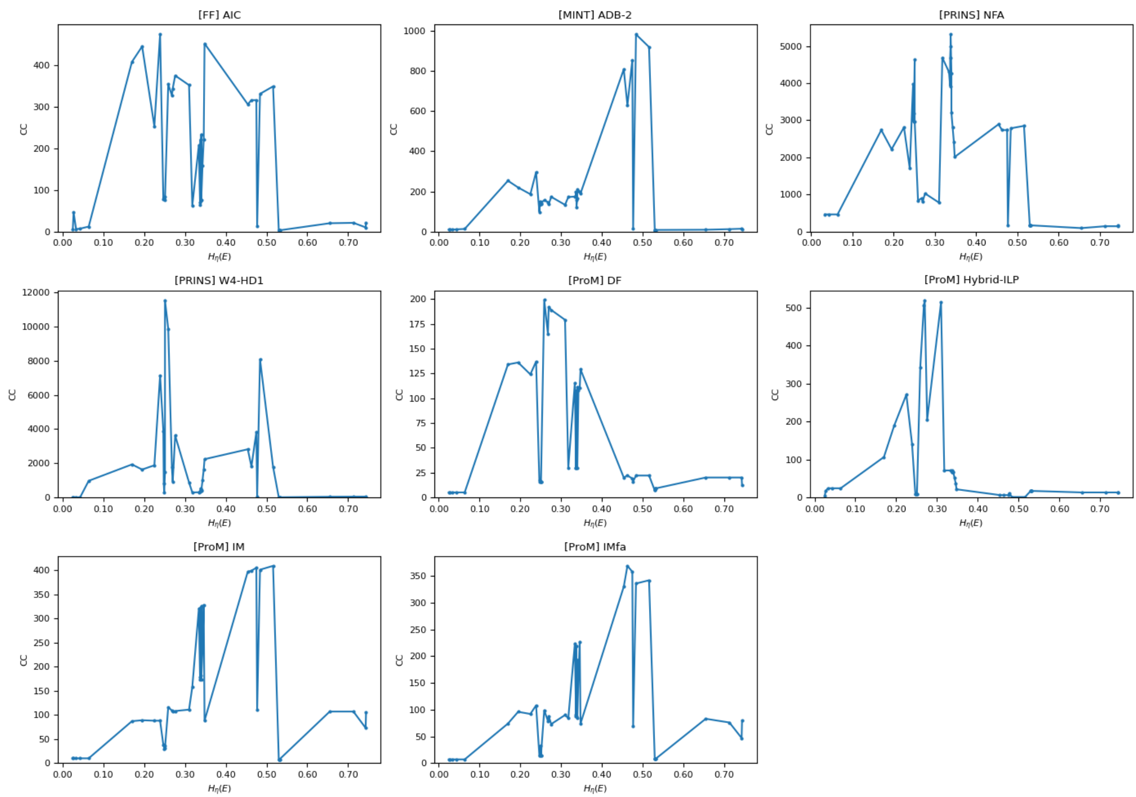


Figure D.11: Cyclomatic Complexity CC vs normalised entropy $H_\eta(e)$

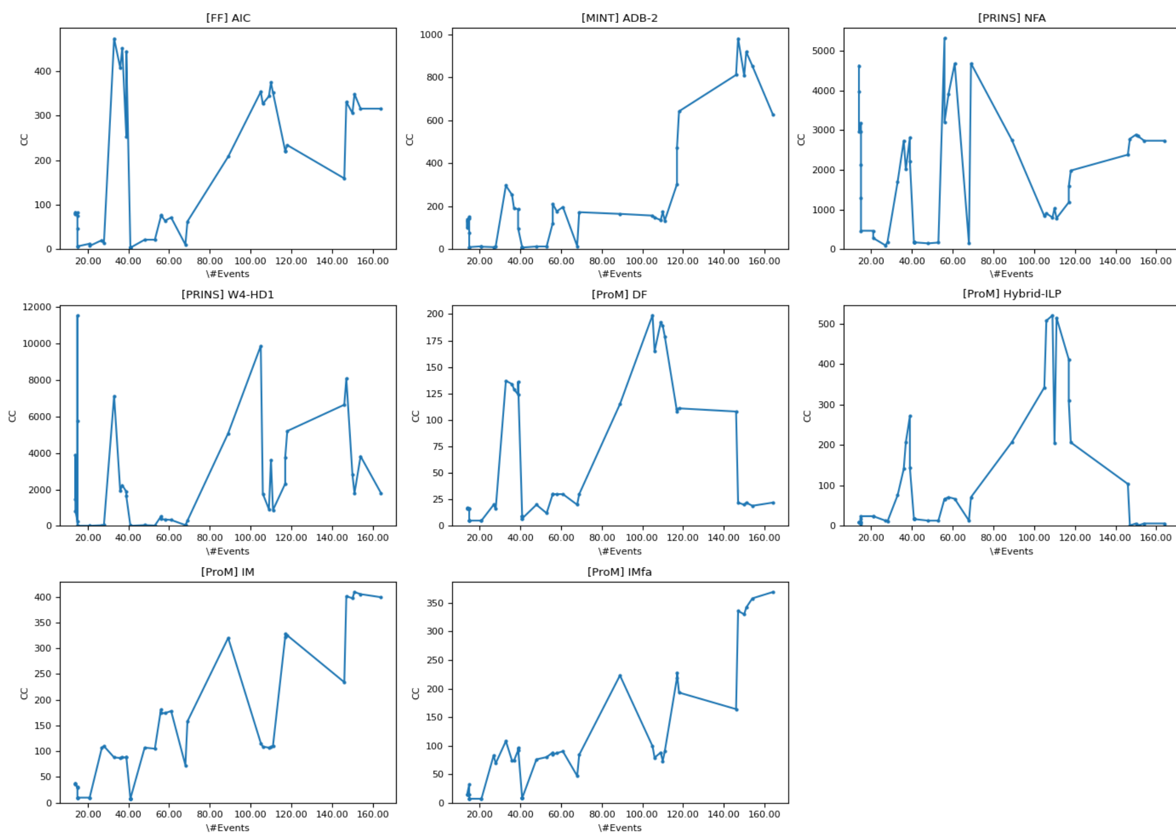
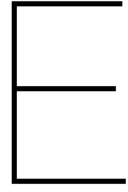
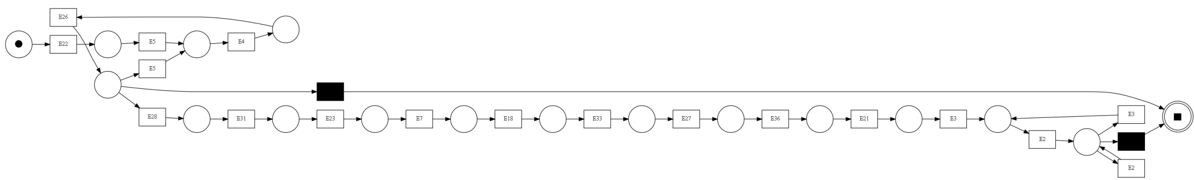


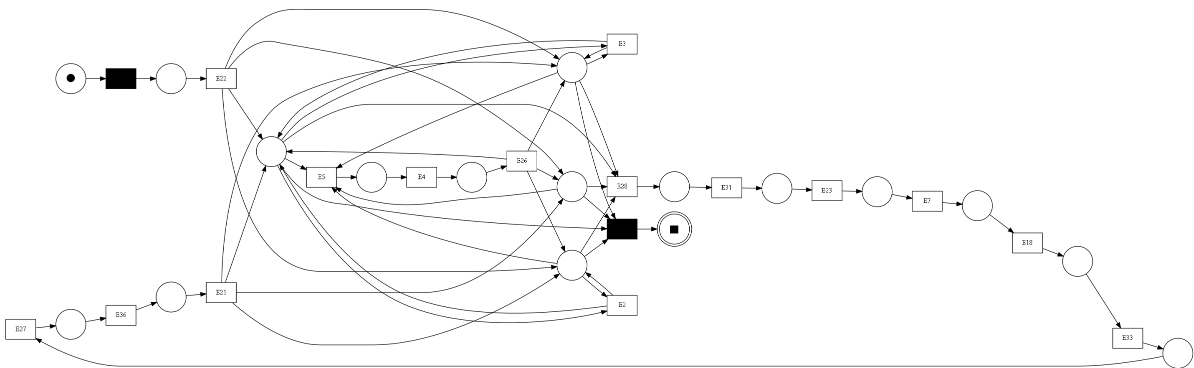
Figure D.12: Cyclomatic Complexity CC vs unique events $\#Events$



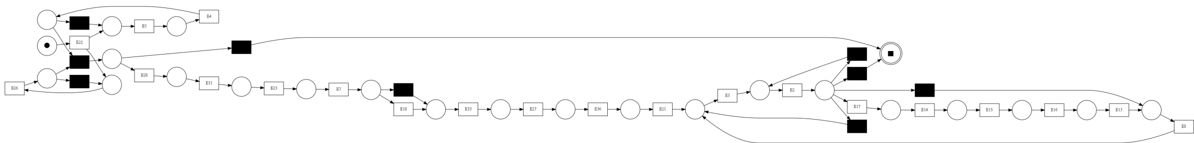
Generated Models



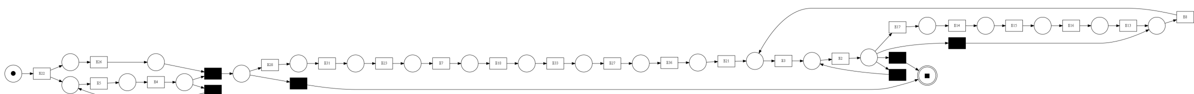
(a) ProM, `DF`, $eCFC$ = 1.08, CC = 5.0



(b) ProM, `Hybrid-ILP`, $eCFC$ = 2.74, CC = 24

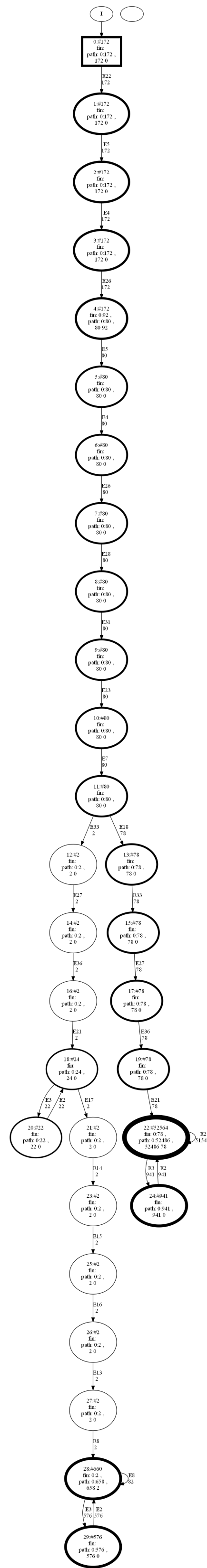


(c) ProM, `IM`, $eCFC$ = 1.13, CC = 10.0

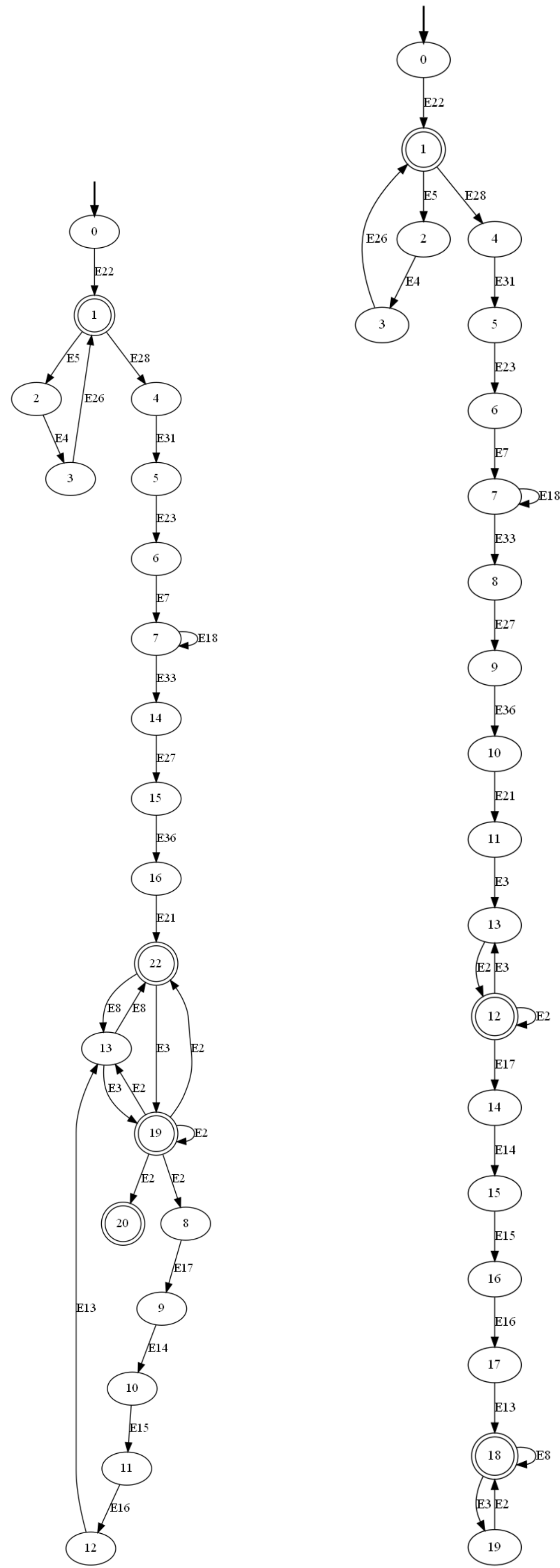


(d) ProM, `IMfa`, $eCFC$ = 1.08, CC = 7.0

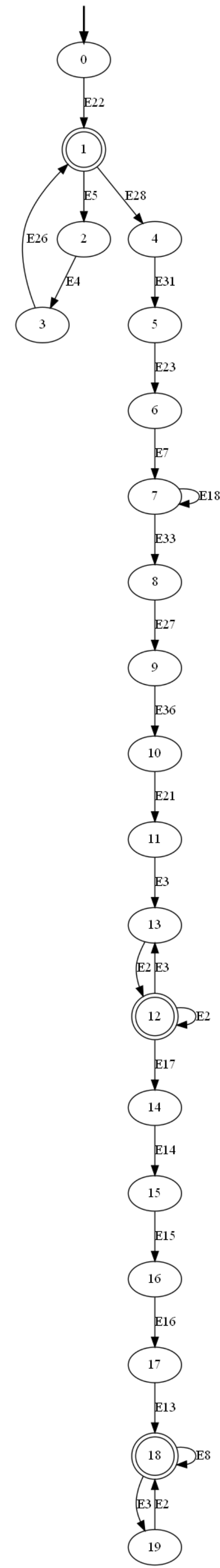
Figure E.1: Spark fold 1 Petri net models



(a) FlexFringe, AIC,
$eCFC = 1.13$, $CC = 6.0$



(b) MINT, ADB-2,
$eCFC = 1.33$, $CC = 9.0$



(c) PRINS, W4-HD1,
$eCFC = 1.24$, $CC = 7.0$

Figure E.2: Spark fold 1 FSM models