Learning Models for Cyber-Physical Systems

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.H.J.M. van Krieken,
volgens besluit van het college van decanen
in het openbaar te verdedigen op woensdag 18 december 2019
om 11:00 uur precies

door

Alexis Linard

geboren op 23 februari 1993
te Troyes (Frankrijk)

Promotor:
Prof. dr. F.W. Vaandrager

Manuscriptcommissie:

Prof. dr. J.H. Geuvers	
Dr. J. Tumova	(Kungliga Tekniska Högskolan, Stockholm, Zweden)
Prof. dr. ir. A.A. Basten	(Technische Universiteit Eindhoven)
Prof. A. Cimatti	(Università di Bologna, Italië)
Prof. dr. C. de la Higuera	(Université de Nantes, Frankrijk)

Dit onderzoek werd mede mogelijk gemaakt door het onderzoekprogramma Robust CPS onder projectnummer 12693, (deels) gefinancierd door de Nederlandse organisatie voor Wetenschappelijk Onderzoek (NWO).

ISBN: 978-94-6323-900-4

Printed by: Gildeprint

Learning Models for Cyber-Physical Systems

Doctoral Thesis

to obtain the degree of doctor
from the Radboud University Nijmegen
on the authority of the Rector Magnificus prof. dr. J.H.J.M. van Krieken,
according to the decision of the Council of Deans
to be defended in public on
Wednesday, December 18, 2019 at 11:00 hours

by

Alexis Linard

born on February 23, 1993
in Troyes, France

Supervisor:
Prof. dr. F.W. Vaandrager

Doctoral Thesis Committee:

Prof. dr. J.H. Geuvers	
Dr. J. Tumova	(Kungliga Tekniska Högskolan, Stockholm, Sweden)
Prof. dr. ir. A.A. Basten	(TU Eindhoven)
Prof. A. Cimatti	(Università di Bologna, Italy)
Prof. dr. C. de la Higuera	(Université de Nantes, France)

This research was supported by the research program Robust CPS with project number 12693, which was (partly) financed by the Netherlands Organization for Scientific Research (NWO).

ISBN: 978-94-6323-900-4

Printed by: Gildeprint

Contents

List of Abbreviations

BN	**B**ayesian **N**etwork
CPS	**C**yber **P**hysical **S**ystem
DFA	**D**eterministic **F**inite **A**utomaton
DBN	**D**ynamic **B**ayesian **N**etwork
DFT	**D**ynamic **F**ault **T**ree
FMS	**F**lexible **M**anufacturing **S**ystem
FT	**F**ault **T**ree
***k*-TSS**	*k*-**T**estable Language in the **S**trict **S**ense
PTA	**P**refix **T**ree **A**cceptor
RPNI	**R**egular **P**ositive **N**egative **I**nference
TA	**T**imed **A**utomaton

To my parents...

Chapter 1

Introduction

Self-driving cars, manufacturing systems, smart power plants, airliners and robots of all kinds often make the headlines. They all have in common to be unavoidable in everyday life since they, in all respects, surround us. With the emergence of computer science and cybernetics since the 1950s, we are witnessing a constant – when not accelerated – complexification of our technological environment. Anyone has in mind the spectacular breakthroughs of the last decades: more and more powerful computers, smartphones, the Internet of Things, big data and artificial intelligence to name just a few examples. This digital revolution also impacted industry, which led to the design of increasingly more sophisticated systems.

The complexity of the design of these systems has become a prominent issue. The advent of Cyber-Physical Systems (CPS) [78, 120] is a striking example of this growth of complexity, but also of the consequent research challenges. CPS comprise systems composed of thousands of software and physical components, the interaction of which is highly subject to situation/environmental changes and, as a consequence, tricky to control. Therefore, they need a constant improvement for their efficiency (e.g. productivity, in the case of manufacturing systems), reliability (i.e. minimizing their downtime) and safety (e.g. safe-by-design, especially needed when interacting with humans).

The complexity of these systems raises new issues. Recent examples brought to light the vulnerability of CPS due to their complexity, and eventually, lack of robust design, reliability analysis, and safety. Whenever the complexity is not well controlled, this can lead to critical failures, not to say life-threatening (see examples of Fig. 1.1). Take, for instance, autonomous and automated cars[1] that are driven by *models* (e.g. artificial neural networks) that command the motor and the wheels, given information returned by a myriad of sensors, observing the environment of the system at all times. These controller models need to be *trained*, i.e. see records of existing situations from which they will learn in order to, once the situation will happen in real life, be able to act accordingly (e.g. choose the right trajectory, brake in front of a crossing pedestrian, etc.). The training task is crucial in such a case. Indeed, it can be the case that facing an unseen situation, a

[1]Several levels of automation are usually considered for self-driving systems [2]. Autonomous cars are said to be cars where the driver is occasionally expected to take over control, whereas, in the case of automated cars, the driver is expected to take over control at any time.

(A) An autonomous car.

(B) An aircraft.

FIGURE 1.1: Two examples of critical Cyber-Physical Systems: an autonomous car (Fig. 1.1a) and an airplane (Fig. 1.1b) which both have been involved in fatal accidents over the past few years.

given model will not be able to make the appropriate decision (for instance, if the model is well trained to detect collisions with human pedestrians, but not with animals). Recent accidents showed such issues, and also the lack of explainable Artificial Intelligence. Indeed, the decisions made by these machine-learned controllers from data is often hard to debug. Problems related to autonomous cars also lead to ethical controversies: in case a collision between the car and a pedestrian would be unavoidable, should the controller hit the pedestrian, or make the car fall by the wayside, risking to kill all the passengers? As a second example, airplanes are now equipped with many components and subsystems aiming at making these safer. However, the recent crashes involving the Maneuvering Characteristics Augmentation System (a.k.a. MCAS) highlighted a safety breach due to poor design. Taking as input the value of only one sensor measuring the angle of attack of the plane, the design choice of the MCAS makes for a single point of failure [12]. The failure of this sensor led to the crash of two aircrafts, while the MCAS was supposed to save lives. Moreover, its failure criticality was neither identified, nor communicated to pilots who could have acted accordingly in case of emergency.

Together, the two examples mentioned above are a source of inspiration for several research problems: first, the need for safety for CPS. Improving safety by design is an exciting research direction since it would avoid safety risks early in the design process of the systems. Second, the need for reliability for CPS: similarly, making robust by design systems would avoid the occurrence of component failures. Most of the CPS now come along with additional sensors, recording the status (downtime, performance...) of their components over time. The data they generate can be adequately employed for several tasks: understand how the system is being used by users of the system and the impact of usage behaviours to the components; figure out under which circumstances (environmental parameters, behavioural usages...) a component is likely to fail; or even retrieve what combinations of component failures can lead to the whole system failure.

In light of the above, we can define the research problems that this thesis will address:

1. how to improve *reliability* and *safety* of CPS, by means of learning accurate reliability models from data?

Luckily, there is a set of techniques that can be applied to CPS: among others, model checking [35], automata learning [70] and reliability engineering [109] can be fruitfully involved, in order to improve the scheduling, control, and reliability of CPS.

All the dramatic examples aforementioned are, fortunately, not covering all types of CPS. The safety consideration for interaction with human beings seems to be a particular case. Other families of CPS count with, however, other considerations such as productivity or cost efficiency. Since studying CPS problems in an abstract and high-level fashion seems rather impractical, we bet on focusing on the practical context of specific CPS to drive things forward. Even though the Dutch industry does not have worldwide leaders in the aeronautic or automotive

sectors, there are currently high-tech companies that develop CPS. Among others, Océ is a leader in industrial printers. Throughout this thesis, we will refer to a case study of industrial printers, like the one depicted in Fig. 1.2a. Despite the considerable difference between autonomous cars, aircrafts and industrial printers, they are all complex CPS. From a computer science perspective, many of the critical issues that play a role in their design are similar. For example, complexity considerations, building reliable and efficient systems using components that may fail, tradeoffs between reliability and costs... are based on common techniques, no matter to what specific CPS they are applied.

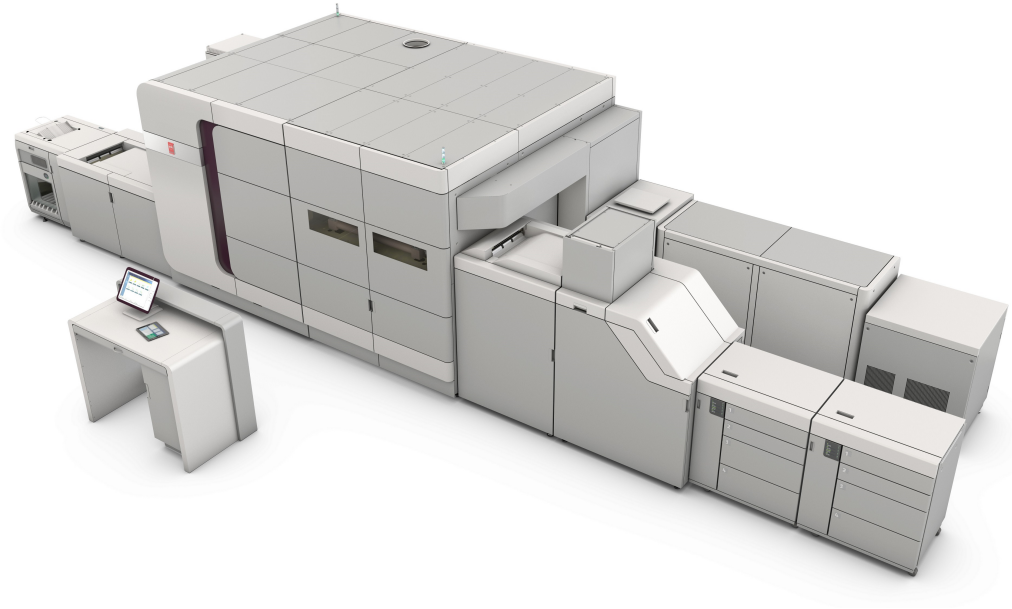
Océ's Large Scale Printers (LSP) are interesting devices. Considered as being a Flexible Manufacturing Systems, they consist of a succession of different components aiming at manufacturing products. They contain a paper path, moving the sheets through different modules aiming at, among others, printing, drying and returning the sheet if this one needs to be printed on both sides. They are also versatile in the types of media that can be used: newspapers, booklets, or merely usual A4 pages. They are high productivity printers, able to print documents in the range of this PhD thesis in less than one minute. However, these devices consist of some critical components, which, in case of failure, can either alter the quality of the print jobs rendered or induce a downtime of the whole system. Such a breakdown is a significant issue for engineers since it inevitably leads to a decrease in the printer's productivity.

The productivity of a printer may not only be affected by the failures inherent to its components. Sub-optimal design choices may also result in low productivity. Take, for instance, a printer specifically designed to print booklets. Since the configuration of the paper path, the heating drum, and even the order in which sheets have to be printed highly depends on the types of jobs being rendered, one can easily imagine that this specific printer will poorly manage to print on newspapers. This does not include, yet, prior information recorded on the functioning, productivity and scheduling of print jobs. Such data nowadays exist and, once again, can be fruitfully employed to solve productivity issues. Hence, with such manufacturing systems, a new research question arises:

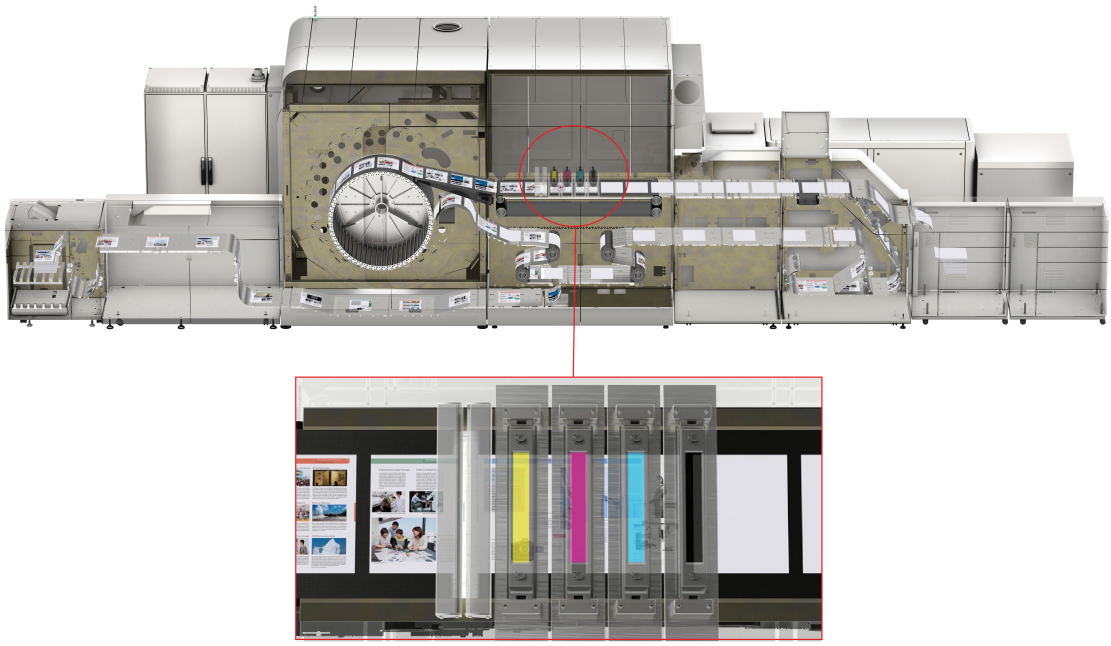
2. how to improve the *efficiency* of CPS, by means of learning models from data?

In the remainder of this thesis, Océ's printers will act as a common thread to all the different chapters, since all experiments – if not, the research content itself – have mainly been inspired by this case study. Indeed, we focused on one critical component of the printer, its printheads as well as its nozzles (tiny components aiming at jetting ink on the paper), as depicted in Fig. 1.2b.

The remainder of this introduction is structured as follows: we first review the main fields this thesis will cover: reliability engineering, learning of models, and productivity issues of CPS. Finally, we will present the outline of the thesis where we expose the contributions in each of the research areas mentioned above.



(A) Overview of the printer.



(B) Detailed view of the printheads.

FIGURE 1.2: Océ's *Varioprint i300* Large Scale Printer.

1.1 Reliability Engineering

Reliability engineering is a field providing methods, tools and techniques to evaluate and mitigate the risks related to complex systems, among others, CPS. As defined in [109], reliability engineering is multi-fold. It aims at:

1. *applying engineering knowledge and specialist techniques to prevent or to reduce the frequency of failures.*
2. *identifying and correcting the causes of failures that do occur.*
3. *determining ways of coping with failures that do occur, if their causes have not been corrected.*
4. *applying methods for estimating the reliability of new designs, and for analyzing reliability data.*

Concerning the first point above, preventive maintenance is often considered, in order to prevent system breakdowns. Current maintenance strategies are often built upon expert knowledge, which uses models that describe the behaviour of the system to maintain. With the growing complexity of systems, human knowledge has been recently recognized as a limiting feature [57, 107]. Naturally, a simple approach for engineers would be to excessively perform maintenance, in the hope of reducing failures at all costs. There is, however, a tradeoff between system's reliability, performed maintenance, and maintenance costs: indeed, one can imagine that the cost of maintenance (both in terms of productivity as financial costs, such as maintenance man-hours, transport costs, storage cost of spare components...) has to be taken into account while designing the maintenance policy. Since the maintenance strategy can influence the reliability of a system, performing smart maintenance is desired. We want at the same time to obtain a more reliable system, but performing maintenance only whenever this is required. However, such an ideal maintenance controller is unavailable. In Chapter 2, we will present a new approach towards adaptive scheduling of maintenance, that is by bringing together machine learning techniques and model checking. This is further discussed in Section 1.4.1.

Concerning points 2 and 4 above, risk assessment [121] is the field which analyzes what the possible failures likely to happen in a system are, what the probable consequences are, and how these failures can be tolerated at the system's level. Risk analysis ensures that systems are safe and reliable. Fault trees [143], such as the one of Fig. 1.3, are a formalism that captures how the failures of components propagate through the system, to lead to the entire system's failure. Among reliability engineering techniques, fault tree analysis [126] is one of the most prominent, to compute dependability metrics such as system reliability and availability, understand how systems can fail, identify the best ways to reduce the risk of system failure, etc. However, fault tree analysis is often based on models, usually built manually by domain experts. Given the complexity of today's systems, fault trees often contain thousands of gates. Hence, their construction

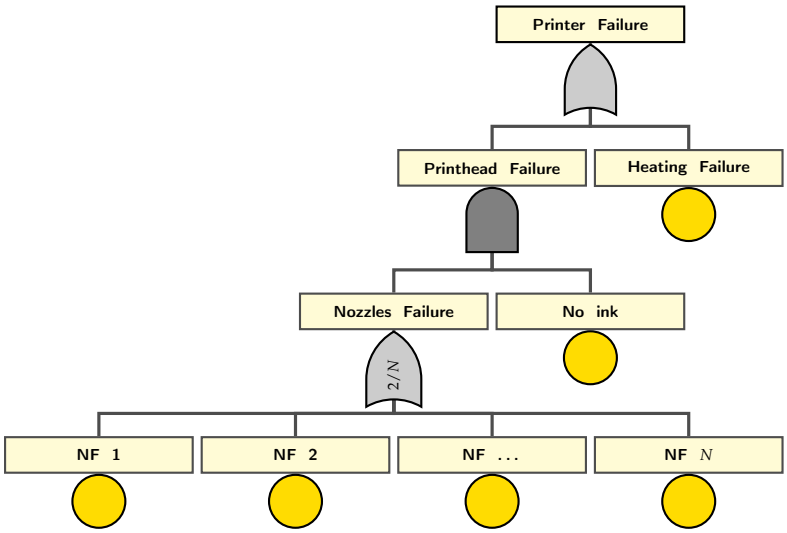


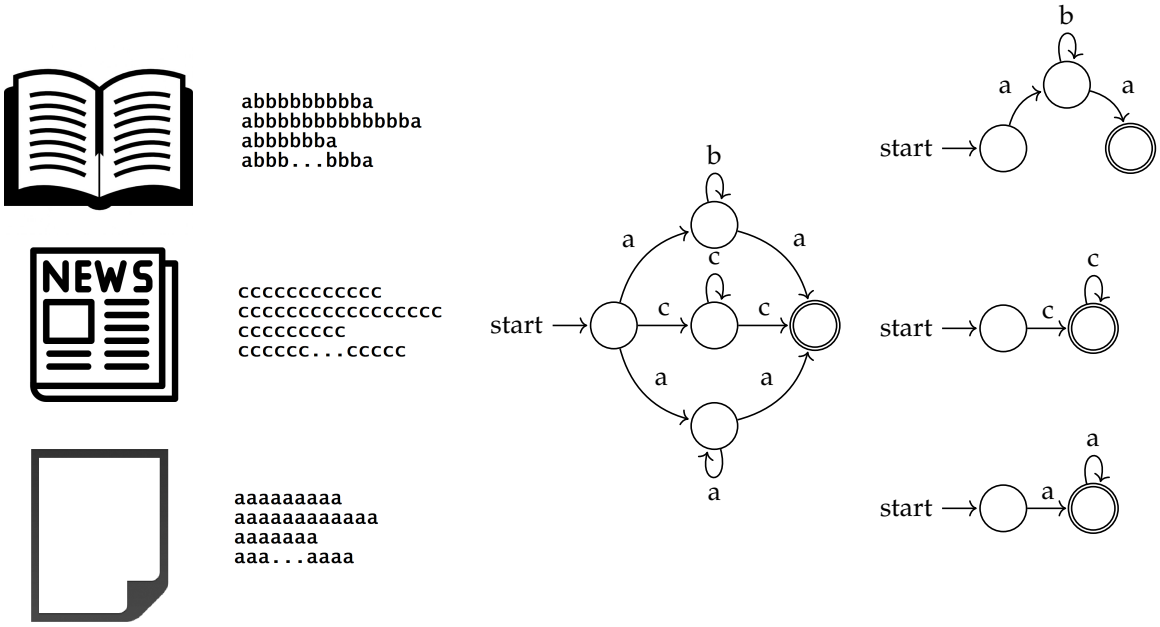
FIGURE 1.3: Example of simplified Fault Tree for a printer. The whole printer fails when either its heating drum fails **or** a printhead fails. A printhead is considered as failing if there is no ink left **and** *at least* 2 of its nozzles are failing.

is an intricate task, and also error-prone, since their soundness and completeness largely depend on domain expertise. In the context of CPS, the status of components is often monitored to get insight into their health. Hence, the sensors measuring this status generate data that can be deployed to learn reliability models. Such data can be crucial for engineers to build fault trees. Recent work focused on learning fault trees from observational data, identifying causalities from data [105]: unfortunately, state-of-the-art algorithms always require the setting of assumptions and lack scalability. In Chapters 3 and 4, we will present two methods performing learning of fault trees from data. They are further discussed in Section 1.4.2.

1.2 Learning of Models

Machine learning [56] comprises a broad set of techniques to learn *models* from data. Here, we make a distinction between two types of models: *white-box* models are models which are transparent, i.e. the structure of which is interpretable. Conversely, *black-box* models are models which take data as input and provides an outcome, but the way the decisions are made is hardly explicable. A current trend is to use neural networks [123], which are, despite their *black-boxness*, robust for tasks such as image recognition or the control of CPS. However, some applications of model learning need interpretability. In this thesis, we advocate the use of more diverse models, and we show that some applications require to be detached from these famous artificial neuron-based models.

A whole area of machine learning is dedicated to the learning of statistical models, which act as a function mapping input data to an output (e.g. the outcome). Also known as supervised learning or classification, the task is to get the most accurate model to fit the input data. They can be applied to many domains, although in this thesis, we will only consider the statistical modelling of predictive tasks. Classifiers are a common factor to several contributions of this thesis: in Chapter 2, a classifier is built in order to reproduce the outcome produced by a costly sensor to query; finally, in Chapters 3 and 4, the method developed is compared to common classifiers, such as logistic regression [16], decision trees [118], the well known neural networks [123], Bayesian classifiers [128] or Support Vector Machines [129].



(A) print jobs and traces. (B) A returned automaton. (C) Desired DFAs.

FIGURE 1.4: An example of print job patterns. In Fig. 1.4a is shown a set of 3 different print jobs (booklet, newspaper and *A4* pages) together with corresponding traces describing jobs in the corresponding categories. Classical automata learning algorithm will return an automaton consistent with all the strings, such as the one shown in Fig. 1.4b. However, the desired learning result is to recover the three different print job patterns separately, as depicted in Fig. 1.4c.

Instead of statistical models, an interesting formalism to consider is one of the state-machines. With an intuitive and universal formalism, they can describe the behaviour of software systems [3], hardware [24], network protocols [55], and are also used in other domains such as linguistics [80] or biology [149]. They contain a set of *states*, and *transitions* between states, that can be taken if certain conditions are met. The state machine can be in exactly one state at a time.

Grammatical inference [70] is the field of learning formal grammars – that can be translated into state-machines – from data. There are two recognized families

of automata learning algorithms: *active* learning algorithms and *passive* learning algorithms. In active learning [6], the learning algorithm interacts with the system, by means of observing outputs of the system given sequences of inputs. Instead, passive learning algorithms [81, 111], given a set of observations describing the behaviour of the system, will construct an automaton from logs. The goal in all cases is to find an automaton (eventually minimal) which is consistent with the observed behaviour of the system. As our case studies were based mainly on logs, we chose to focus on passive learning algorithms for learning Deterministic Finite Automata (DFA). Considering the set of words of Fig. 1.4a, usual passive learning algorithms will return an automaton describing the language (or grammar) generating these words. However, this particular set of traces, which describe print jobs sent to Large Scale Printers, contain several clusters: print jobs of the booklet family, of the newspaper family, and of the A4 documents. A desirable result would be then, not to cover all the words through a single language (Fig. 1.4b), but instead to learn a language for each of the three different families of print jobs (Fig. 1.4c). In Chapters 5 and 6, we will present two methods for learning unions of languages. They are discussed in Section 1.4.3.

1.3 Efficiency of CPS

The efficiency of CPS is multi-fold: the first range of productivity considerations covers the ability of a system to perform optimally. Indeed, an important consideration is to reach the optimal level of functioning at which the system is designed to perform. Therefore, optimization of the tasks CPS have to execute is crucial to ensure the cost efficiency of the systems. Throughout the thesis, the Large Scale Printer example shows how tricky these considerations can be. One complicated issue is one of paper path scheduling. A paper path consists of a Paper Input Station, an Image Transfer Station, a Re-entrant Loop with a Turn Station, and a Finish Station. A scheduler must choose in which order the different sheets need to pass the Image Transfer Station. The scheduling decision is sensitive to the nature of a job: that is, a booklet, a newspaper, and an A4 document will be scheduled in different ways. The scheduling decisions are, therefore, critical for optimizing the printer's productivity. To control the decisions, a ranking metric is used to schedule the paper passes. However, the ranking metrics are often designed by domain experts, which reduces the search space of the schedulers to expert knowledge. Data being available on the different print jobs being usually rendered, as well as the nature and productivity outcome of different jobs given specific printer settings, machine learning seems to be, once again, the solution to this optimization problem. In Chapter 7, we will present a machine learning method to improve the productivity of Large Scale Printers, reusing results of Chapters 5 and 6. They are further discussed in Section 1.4.4.

Another productivity consideration is the ability of a system to maintain an acceptable productivity level given failures and downtime. We show in the thesis that risk management may also lead to productivity issues: when too many maintenance actions are performed, the productivity may decrease. Yet, finding

the optimal threshold between maintenance and productivity matters for another optimization problem. In all chapters of this thesis, we will show that the different contributions bridge the gap between the different challenges around CPS, so that, *in fine*, the productivity is enhanced:

- Chapter 2 brings together machine learning and model checking to improve the reliability of CPS at low costs.
- Chapters 3 and 4 use machine learning of reliability models to improve the safety of CPS and, in the end, reduce their downtime and increase their productivity.
- Chapters 5 and 6 use model learning, more precisely, automata learning, the results of which are used in Chapter 7 to boost their productivity.

1.4 Outline of the thesis

The chapters of this thesis consist of research papers the author contributed to. They span the research areas of efficiency, reliability and safety of CPS, as shown in Table 1.1. Further, they can be grouped into the four following themes: learning maintenance policies for CPS, learning of fault trees for CPS, learning unions of models for CPS and learning of heuristic schedulers for CPS.

Efficiency	**Reliability & Safety**
Learning of Maintenance Policies for CPS (Chapter 2)	
	Learning of Fault Trees for CPS (Chapters 3 & 4)
Learning Unions of Models for CPS (Chapters 5 & 6)	
Learning of Heuristic Schedulers for CPS (Chapter 7)	

TABLE 1.1: CPS research areas spanned by the different contributions of the thesis.

1.4.1 Learning Maintenance Policies for CPS

Scheduling and Control of CPS are becoming increasingly complex and multi-fold: engineers are interested in accurate controllers to deal with component's interaction, and also for failure detection and scheduling component replacements. Maintenance policies are usually time-monitoring based, which can lead to sub-optimal scheduling of maintenance. In this work, we improved maintenance scheduling of an industrial printer, through monitoring in real-time the critical components and dynamically adjusting the optimal time between maintenance actions. We considered a case study of Large Scale Printers, and more particularly of one of its components: the printing nozzles. These are designed for jetting ink on paper according to specifications depending on, e.g. the type of paper being used, or the image that needs to be transferred on the paper. During the operation of Large Scale Printers, nozzles can behave inadequately for the demanded task, e.g. by jetting incorrect amounts of ink or jetting in a wrong direction. These "failures" can be solved by performing automated maintenance actions, including, for example, different types of cleaning actions. In this context, determining the appropriate moment to execute each nozzle related maintenance action is crucial to achieving a proper balance between productivity, reliability and, of course, final product quality.

We used machine learning and model checking to obtain an adaptive maintenance controller, which we applied to Océ's Large Scale Printers. The assets of the machine learning techniques we used rely on the learning of models, e.g. classifiers, from data. They classify instances into a set of possible classes (here, *failure* or *non-failure*), according to the values of the attributes of each data record. In practice, they are queried with data records which are not labelled, that is, the class of which is unknown. A feature of Large Scale Printers is also the fact that they record large amounts of data about the state of their components over time, hence the additional motivation to apply machine learning to our case. In this work, we use the classifier as a tool to reproduce the outcome of a costly sensor measuring whether a printing nozzle is failing or not.

Maintenance actions are often scheduled with fixed intervals that are suboptimal and implemented to the detriment of productivity and efficiency. The desired result is, therefore, to perform maintenance only when necessary. The goal is then to deviate from the "typical" triggering of maintenance substantially, that is by postponing maintenance when no failure is likely to happen, or to advance the maintenance actions in case a failure is about to occur. The maintenance policy of a system can, therefore, be represented intuitively through a specific type of state machine: a timed automaton [5], which is a finite-state machine extended with a finite set of real-valued clocks constraints. Its states represent the different maintenance actions and its transitions when a given maintenance should be triggered. Interestingly, some model checking tools take timed automata as input. *Uppaal* [14] is such a toolbox for verification of real-time systems represented by one or more timed automata extended with integer variables, data types and channel synchronization. The relevance of using the model checking tool *Uppaal* in our case lies in the possibility of running simulations of the system specified

as a set of probabilistic timed automata. In practice, by modelling the printer usage (if the printer is idle or being used), the maintenance scheduler (how and when maintenance actions are performed) and the scheduler updater (calling the failure classifier with a given frequency) into *Uppaal*, we could make simulations and evaluate the benefits of our dynamic scheduler. We can also find the relevant parameters of the dynamic maintenance scheduler, such as, among others, how much to advance or postpone the maintenance triggering each time the classifier is queried.

We claim that maintenance actions are frequently scheduled with intervals that are suboptimal. We also note during our experiments, that the price to pay in order to do less maintenance is a slight increase of the expected failure rate.

This research is detailed in:

Chapter 2: Towards Adaptive Scheduling of Maintenance for Cyber-Physical Systems

This chapter is based on the following publication:

Linard, A., Bueno, M. L. "Towards Adaptive Scheduling of Maintenance for Cyber-Physical Systems". 2016. In *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, 134-150*.

The author has presented this work at the 7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2016) in Corfu, Greece, on Monday, October 10 2016.

1.4.2 Learning of Fault Trees for CPS

CPS come with increasingly intricate architectures and failure modes. Usually, expert knowledge is detailed on the functioning of individual components. However, the understanding of their interaction is far from simple, since it may also rely on external factors, such as environmental parameters or user interaction. For these reasons, obtaining accurate system reliability models becomes a hard task. In this part, we consider the context of the Internet-of-Things [9], where systems tend to be continuously monitored via advanced sensor systems. Hence, data is available on the components' failure behaviour. We use this data to learn fault trees, a widely spread formalism in reliability engineering, which captures the failure behaviour of components and their propagation through an entire system.

Firstly, we focused on fault tree generation from data, using an evolutionary algorithm. This family of algorithms approximate stochastic learning by mimicking biological evolution. Being an iterative process, each stage of the algorithm keeps a population of candidate fault trees. A generation of new fault trees is then generated by reproduction (e.g., adding or deleting gates to the fault trees), crossover (e.g. swapping fault tree branches), and mutation (e.g. changing an

AND gate into an OR gate). Finally, we select the new population among the best-fitted fault trees, i.e. with the best fit to the input data.

Secondly, we focused on fault tree generation from data, using Bayesian inference. Bayesian networks [87] are standard probabilistic graphical models which can be learnt from data. They are widely used in industry, as well as in the health domain as a diagnosis tool. Bayesian Networks are also used in dependability analysis of systems, being an interesting formalism for the generation of qualitative and diagnosis measurements. Hence, there exist straightforward relations between fault trees and Bayesian networks [21], since once may want to translate a fault tree into a Bayesian network to obtain such metrics. The first step is, therefore, to infer a Bayesian Network from data. Prior assumptions on the problem to solve can be integrated into the learning algorithm through *blacklists*, which is done by forbidding some edges between variables in the final structure of the Bayesian Network. The second and last step is to translate the obtained Bayesian Network into a fault tree. To that end, we apply a set of rules to apply to the structure of Bayesian Network, as well as to the coming along conditional property tables. Among others, the conditional property tables help to define the nature of the fault tree gates, i.e. whether it is an AND or an OR gate.

We applied this research on Large Scale Printers to ensure the relevance of our methods, as well as on a benchmark of fault trees. In both cases, we could show that our methods are efficient and accurate. We also developed an alternative to the basic algorithms, that is by taking an additional input: a *skeleton* fault tree. This skeleton is meant to be expert knowledge to be aggregated to the data-driven approaches. Indeed, we have observed that, in practice, experts already have an idea of the fault tree structure, but only need to verify or complete the entire fault tree model thanks to available data. To that end, we modified the first approach, to make sure that given skeletons remain unmodified in each mutated fault tree; and the second approach, defining *whitelists*, which are edges known to be present during the Bayesian Network structure learning.

This research is detailed in:

Chapter 3: Fault Trees from Data: Efficient Learning with an Evolutionary Algorithm

This chapter is based on the following paper:

Linard, A., Bucur, D., Stoelinga, M. "Fault Trees from Data: Efficient Learning with an Evolutionary Algorithm". 2019. In *Proceedings of the Symposium on Dependable Software Engineering: Theories, Tools, and Applications, SETTA 2019. 27 - 29 November 2019, Shanghai, China.*

The author has presented this work at the Symposium on Dependable Software Engineering: Theories, Tools, and Applications (SETTA 2019) in Shanghai, China, on Thursday, November 28 2019.

Chapter 4: Learning Fault Trees through Bayesian Networks
This chapter is based on the following publication:

Linard, A., Bueno, M., Bucur, D., Stoelinga, M. "Induction of Fault Trees through Bayesian Networks". 2019. In *Proceedings of the 29th European Safety and Reliability Conference, ESREL 2019.*

The author has presented this work at the 29th European Safety and Reliability Conference (ESREL 2019) in Hanover, Germany, on Monday, September 23 2019.

1.4.3 Learning Unions of Models for CPS

Learning of models for CPS is a common task, as well for software as for hardware. When data is available prior to the learning task, *passive learning* algorithms can, given traces, return a model. When data is not available, *active learning* algorithms will query the system to be learned, and observing the output of these queries, will incrementally build a hypothesis model of it. A classical problem in passive learning is to identify a language from a set of examples. However, the main assumption when learning the model is that the traces describe one unique behaviour. In this part, we address the problem of identifying a union of languages from examples that belong to several *different* unknown languages. This is based on the novel assumption that the traces may represent distinct behaviours, i.e. the traces belong to different clusters of actions. An additional motivation is that decomposing a language into smaller pieces that are easier to represent should make learning easier than aiming for a too generalized language. Applied to Large Scale Printers, and especially to the print jobs being sent to the printers, the discovery of a language describing the structure of print jobs is impractical, but the decomposition into similar job structures of print jobs sharing similarities is a fruitful result. The smaller languages can therefore illustrate more specific behaviours, such as printing newspaper, booklets, envelops...

Firstly, we considered the class of k-testable languages in the strict sense [99]. They are defined by a set of allowed prefixes, infixes (sub-strings) and suffixes that words in the language may contain. k-testable languages have been fruitfully applied to a broad range of domains, such as XML files [17], DNA analysis [44] or Natural Language Processing [122]. They are also expressive enough to represent the print jobs sent to Large Scale Printers. One reason for their popularity is that, unlike regular languages, k-testable languages are learnable in the limit from text. In other words, there exists an algorithm that, after reading a given number of strings belonging to the language, will converge to the target k-testable language in a finite number of iterations. Intuitively, it means that the target language is identified whenever all different allowed prefixes, infixes (sub-strings) and suffixes have been read. The first contribution we bring to the study of k-testable languages is a Galois connection between the lattice of all languages over alphabet Σ and the lattice of k-testable languages over Σ. We used this Galois connection to prove the learnability in the limit of unions of k-testable languages, as well as to derive an algorithm to infer unions of k-testable languages. As we

show, this naive algorithm may not be optimal. Therefore, we develop a hierarchical clustering algorithm, based on a distance between k-testable languages.

Secondly, we considered the problem of learning unions of regular languages. Unlike k-testable languages, unions of regular languages are not learnable in the limit. Indeed, deciding whether any two strings should be in one of the languages in the union, rather than in another, are matters of heuristic choices. Thus, we defined two techniques for learning unions of regular languages from positive examples, and two techniques for learning unions of regular languages from positive *and* negative examples:

1. *Clustering by Compression*: this method clusters strings based on their *compressed distance* [41]. Since compression algorithms usually try to find repetitions in strings to *pump* them, strings sharing the same repetitions are likely to have a small *compressed distance*. We hope then to cluster structurally similar strings together, and to learn a language for each of the identified clusters.

2. *Identification of Tandem Repeats*: this method first computes the pumping decompositions of the strings by computing tandem repeats [136]. Tandem repeats are side-by-side repetitions of substrings occurring twice or more. Assuming that two strings belong to the same language if they contain a similar prefix, tandem repeat, and suffix, we can then perform a hierarchical clustering of languages, by successively making generalizations up to Σ^*.

3. *State-merging for Unions of Languages*: a common technique in grammatical inference is to perform state-merging given a prefix-tree acceptor of all strings in the dataset. Here, we modify RPNI [111] to detect iterations of state-merging involving a large number of states, in the hope of capturing semantically distinct behaviours, that would result in a union of languages.

4. *Evolutionary Algorithm for Learning Unions of Languages*: finally, our last method is based on an multi-objective evolutionary algorithm [47], trying to optimize the number of languages in the union (which is given as input) as well as the accuracy of the union of languages (that is, making sure that all strings in the positive sample are accepted by a language, and that all the languages reject all strings in the negative sample in the union).

We applied our results to sets of print jobs, which describe distinct customer usages. The discovery of print job patterns is beneficial for engineers designing the printers: indeed, some printer related settings need to be tuned depending on the type of print job an instance of the printer is likely to print. The results of our algorithms have been used in Chapter 7. This has been used to, amongst others, increase the productivity of certain print jobs. We could show that the prior identification of print job patterns can be used as extra knowledge to improve a paper path scheduler.

This research is detailed in:

Chapter 5: Learning Unions of k-Testable Languages
This chapter is based on the following publication:

Linard, A., de la Higuera, C., Vaandrager, F. "Learning Unions of k-Testable Languages". 2019. In *Proceedings of the 13th International Conference on Language and Automata Theory and Applications, LATA 2019*

The author has presented this work at the 13th International Conference on Language and Automata Theory and Applications (LATA 2019) in St Petersburg, Russia, on Thursday, March 28 2019.

Chapter 6: Towards Learning Unions of Regular Languages
This chapter is based on the following publications:

Linard, A. "Learning Several Languages from Labeled Strings: State Merging and Evolutionary Approaches". 2018. 2nd workshop on Learning and Automata (LearnAut 2018). *arXiv preprint:* arXiv:1806.01630

Linard, A., Smetsers, R., Vaandrager, F., Waqas, U., van Pinxten, J., Verwer, S. "Learning Pairwise Disjoint Simple Languages from Positive Examples". 2017. 1st workshop on Learning and Automata (LearnAut 2017). *arXiv preprint:* arXiv:1706.01663

The author has presented these papers at the 1st workshop on Learning and Automata (LearnAut) in Reykjavik, Iceland, on Monday, June 19 2017, as well as at the 2nd workshop on Learning and Automata (LearnAut) in Oxford, the UK, on Friday, July 13 2018.

1.4.4 Learning of Heuristic Schedulers for CPS

Scheduling of processes is required to achieve high productivity of CPS. In the case of Large Scale Printers printing hundreds of sheets per minute, proper scheduling of print jobs is crucial. Since Large Scale Printers enable double-sided printing, the re-entrant feature of their paper path makes the scheduling of print jobs a hard task. Scheduling can be done thanks to expert knowledge or using a heuristic scheduler. Here, we investigated the discovery of new heuristics to improve the scheduling and performance of Large Scale Printers.

Solving combinatorial problems about paper path scheduling can be difficult, and online solutions are generally based on heuristics. These heuristics are, in practice, built upon expert knowledge. However, the acquisition of such human expertise is costly (over the years) and may be biased towards particular solutions. Therefore, there has been an increase in attention for automatically generating new heuristics for search problems. These techniques, called hyper-heuristics, are amongst others based on genetic programming [33]. In the case of

Large Scale Printers, the scheduling freedom consists of choosing the ordering of the products at the re-entrant machine. Such ordering is decided upon ranking metrics. We developed four different methods based on genetic programming approaches, which iteratively optimizes a population of candidate ranking metrics, by randomly mutating and combining solutions from a population, such that the fitness of the best solution (i.e. the printing time rendered by the metric) per population improves in time. The first method aims at solving a given print job. The second aims at solving a group of print jobs (i.e. jobs falling into the same category). The third and the fourth aims at respectively solving a given print job and a group of print jobs by running two genetic programs in a row.

From a practical perspective, print jobs can be grouped into different categories, based on structural similarities. Take, for instance, the print job patterns that Chapters 5 and 6 aim at finding. Then, the number of pages only differs between print jobs. Finding ranking metrics that optimize for such families can yield improvements for the online scheduler. We could show that our different methods achieved a significant reduction in the jobs' printing times, which ensures their relevance.

This research is detailed in:

Chapter 7: An Application of Hyper-Heuristics to Flexible Manufacturing Systems

This chapter is based on the following publication:

Linard, A., van Pinxten, J. "An Application of Hyper-Heuristics to Flexible Manufacturing Systems". 2019. In *Proceedings of the 22nd Euromicro Conference on Digital System Design, DSD 2019*.

The author has presented this work at the 22nd Euromicro Conference on Digital System Design (DSD 2019) in Kallithea, Greece, on Wednesday, August 28 2019.

Finally, Chapter 8 is dedicated to the conclusion of this thesis, and a discussion on future research challenges.

Chapter 2

Towards Adaptive Scheduling of Maintenance for Cyber-Physical Systems

Scheduling and control of Cyber-Physical Systems (CPS) are becoming increasingly complex, requiring the development of new techniques that can effectively lead to their advancement. This is also the case for failure detection and scheduling component replacements. The large number of factors that influence how failures occur during operation of a CPS may result in maintenance policies that are time-monitoring based, which can lead to suboptimal scheduling of maintenance. This chapter investigates how to improve maintenance scheduling of such complex embedded systems, by means of monitoring in real-time the critical components and dynamically adjusting the optimal time between maintenance actions. The proposed technique relies on machine learning classification models in order to classify component failure cases vs non-failure cases, and on real-time updating of the maintenance policy of the sub-system in question. We use model checking to simulate the benefits of our technique. The results obtained from the domain of printers show that a model that is responsive to the environmental changes can enable consumable savings while keeping the same product quality, and thus be relevant for industrial purposes.

2.1 Introduction

Due to the growing complexity of Cyber-Physical Systems [37, 82], many techniques have been proposed to improve failure detection and scheduling component replacement [68, 97]. Indeed, new needs in terms of reliability and safety have appeared with the new applications of such systems. That is the reason why leading-edge technology manufacturers seek to design more robust and reliable systems [48]. Currently, a major issue in many industrial settings is how to correlate failure occurrences and the maintenance actions performed in order to prevent breakdowns. Modelling the failure behaviour of many components in advance is an intricate task. Indeed, we claim that maintenance actions are frequently scheduled with fixed intervals that are suboptimal and implemented to the detriment of productivity and efficiency.

Exclusively relying on experts to build models that can describe the behaviour of machines has been recently recognized as a limiting feature [57, 107]. Therefore, the use of machine learning techniques to construct such models has been investigated [8, 36, 65, 76]. However, using such techniques in order to update the maintenance scheduling of a CPS in real time has so far not been explored. The main difficulties, in this case, relate to finding an appropriate predictive model and then defining a procedure for updating the timing conditions. Predictive models can be used instead of costly sensors intended to provide information about the state of the machine at any moment, which is an additional reason why we introduce machine learning techniques to maintenance scheduling. The ultimate goal would be to develop embedded systems capable of dynamically scheduling their own maintenance. To that end, we aim to define a procedure for updating in real time when maintenance should be performed. Our work is based on an experiment carried out in partnership with industry, specifically in the domain of printers. In addition, we consider the scope of *automatic* maintenance, where the intervention of human beings is no longer needed.

In this study, we investigate to what extent machine learning can help to improve the fixed maintenance scheduling of complex embedded systems. The contributions of this chapter are as follows. First, we propose using machine learning techniques, in which the embedded system learns to distinguish between failure vs non-failure cases using data related to critical components of the CPS. This can be done by monitoring critical components in real time. Next, we propose an algorithm to adjust the timing of maintenance actions dynamically. This algorithm uses timed automata [5], which is the formalism used to model the maintenance policy. Indeed, timed automata can provide an intuitive representation of the maintenance policy, and its real-time update is proceeded by using information on the overall printer at any moment. Thus, our main contributions are (a) the use of decisions from a data-driven model to dynamically schedule maintenance, and (b) the use of timed automata to formally describe and analyze the proposed algorithm. Naturally, we only select relevant features to determine if the printer is working correctly – that is to say if we can schedule the maintenance actions later – or not. To that end, we consider a set of realistic, industry-based scenarios and simulations to provide evidence that a reduced amount of maintenance can be done while achieving similar product quality [30, 35]. The considered scenarios have been implemented in *Uppaal* [14], a model checking tool, which is another relevant practical-oriented contribution of this chapter.

The remainder of this chapter is organized as follows. In Section 2.2, we present the industrial problem that motivated our approach. In Section 2.3, we define the key concepts associated with model-based scheduling and classification techniques used to separate the failure from the non-failure cases of a Cyber-Physical System, as well as discuss the related literature. In Section 2.4, we explain our approach to updating when to trigger maintenance and the experiments done, using a model-checking tool and data about Large Scale Printers. Finally, we discuss the results.

2.2 Case Study

Large Scale Printers are Cyber-Physical Systems made of a large number of complex components, the interaction of which is often challenging to understand. Among their main components are the printheads, which are composed of thousands of printing nozzles. These are designed for jetting ink on paper according to specifications concerning, for example, jetting velocity and direction. During the operation of these industrial printers, nozzles can behave inadequately for the demanded task, e.g. by jetting incorrect amounts of ink or jetting in a wrong direction. If that is the case, a nozzle is considered to be failing.

Failing nozzles can be repaired by performing one or more maintenance actions, including, for example, different types of cleaning actions. The maintenance actions are executed automatically, e.g. the printer cleans its own nozzles. In this context, determining the appropriate moment to execute each nozzle-related maintenance action is crucial to achieving a proper balance of conflicting objectives, such as productivity, machine lifetime, and final product quality. However, the number of individual nozzles, their physical architecture, the substantial number of variables that can potentially be correlated to them and the definition of printing quality, make particularly difficult to manually construct models that express all the potentially relevant correlations among these variables and nozzles. Ultimately, this creates a challenge when designing maintenance policies, since they are intended to be either too conservative or tolerant. Otherwise, one or more of the mentioned requirements could be severely degraded.

A solution that is sometimes used to construct a policy consists of performing a large set of tests involving different usages of the CPS, aiming to derive time-based maintenance policies. These policies are based on monitoring. Hence they do not suffer so much from the drawbacks of fixed-time-based strategies [68]. However, these policies rely on time counters that are not directly related to the state of machine parts, e.g. the time elapsed since the last finished printing task and the period that the printer stands idle. This implies that the only failure behaviours that can be explicitly captured by these rules are those that were previously seen during the tests, usually only accounting for a fraction of all possible machine statuses. Hence, unseen failure behaviour cannot be appropriately handled by such maintenance strategies, which is likely since these machines can be used with a wide range of parameter combinations. In other words, such policies are prone to perform blindly on at least some situations, which can lead to too many or too few maintenance actions.

Nowadays, industrial printers record large amounts of data about the state of their components over time, so a data-driven approach seems feasible. A data-driven approach can provide evidence that helps to decide on whether the current time parameters are adequate or not, given the current state of the CPS. In order to represent time parameters and system states, a state machine appears to be an appropriate formalism. In this study, we show how to dynamically update the parameters of a maintenance scheduler, which is based on timed state

machines. This formalism is used in order to capture the intuition that the CPS moves between different states during its operation, in a time-dependent manner. A significant advantage of combining a statistical approach and state machines is that the real-time updating of timed parameters requires no human intervention since correlations between potentially relevant variables can be learned algorithmically. In addition, the involvement of a failure behaviour model is substantiated in our case, since there is no sensor available to directly provide, at any moment, information on the state of the nozzles. That is the reason why such a model could provide the desired outcome whenever required.

2.3 Background

In this section, we present the different concepts on which our definition of an adaptive scheduler is based. First, we describe how to model and verify maintenance schedulers. Then, we discuss the machine learning techniques used in our chapter and to what extent the need for them is relevant for real-time updating of scheduling. Finally, we discuss the related literature.

2.3.1 Modeling Maintenance Strategies

State machines are abstract machines with a wide range of applications such as in process modelling, software checking and pattern matching. They are composed of a set of states and transitions between states. They can be used in our industrial case study, that is to say, modelling maintenance actions schedules of CPS, since it is possible to gather as many states as different maintenance actions plus one or more states representing when no maintenance action (MA) is being performed in the CPS. Transitions between the states would take place when a given maintenance is started and completed [1]. As shown in Figure 2.1, the maintenance policy of a system can be represented intuitively by means of a specific type of state machine: a timed automaton (TA).

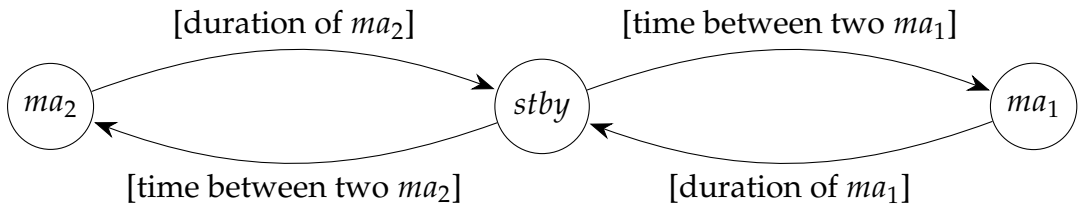


FIGURE 2.1: Maintenance policy of a component represented by a simplified TA.

A timed automaton [5] is a finite-state machine extended with a finite set of real-valued clocks constraints. During the execution of a timed automaton, clock values increase at the same speed. A timed automaton also has clock guards, that enable or disable transitions and constrain the possible behaviours of the automaton. Furthermore, when a transition occurs, clocks can be reset.

The particular TA presented in Figure 2.1 is a way of modelling a given maintenance strategy. In our case, we assume that maintenance actions are executed sequentially. In order to establish that the derived maintenance strategy achieves an optimal trade-off, many techniques exist, including model-checking of real-time systems [35]. We used the tool *Uppaal* [14] to evaluate time-monitoring-based maintenance strategies. More details about how this tool was used are presented in Section 2.4.2.

2.3.2 Classification Techniques

This study relies on the use of classification techniques [56], also known as supervised learning, belonging to the field of Machine Learning. These techniques consist of learning models from data. They are designed to classify instances into a set of possible classes, according to the values of the attributes of each instance. To that end, we used Weka [148], a suite of implemented learning algorithms. Among the possible classifiers that can be used for the classification task, we considered in particular: Bayesian networks, naive Bayes classifiers, decision trees, random forests and neural networks.

The main reason for the choice of the classifiers listed above is related to our case of study. Indeed, most of them (excepted neural networks) can be considered as *white box* classifiers, in the sense that it is easy to interpret their provided outcomes. This is particularly important in the context of industrial cases because such a readable model can be more insightful than a *black box* oracle.

In order to learn a classifier from log data, labelled data from each possible class is needed. In the case of learning the failure behaviour of the CPS of interest (i.e. nozzles of Large Scale Printers), we assume that nozzles are either failing or not failing. Thus, each instance is composed of several features corresponding to relevant machine components and the class feature indicating the occurrence of a failure or not.

actual→ / predicted↓	f	!f
f	A	B
!f	C	D

$$P_f = \frac{A}{A+C} \quad R_f = \frac{A}{A+B} \quad P_{!f} = \frac{B}{B+D} \quad R_{!f} = \frac{B}{C+D}$$

FIGURE 2.2: Confusion matrix and evaluation criteria for the classes *failure* (denoted by f) and *non-failure* (denoted by !f).

In machine learning, an important goal when learning classifiers is that of adequately generalizing to new data. To meet this goal, overfitted models should be detected since they tend to perform very well on the data used during learning. This issue is often dealt with by using the n fold cross-validation evaluation. Cross-validation with n folds consists of first dividing the dataset into n disjoint

sets, then learning the model on the $n-1$ first folds and evaluating the learned model on the last fold. Then, learning is done on folds 2 to n, and evaluation is done on fold 1, and so on. The part of the data used to learn a model is usually called training data, while the part used to evaluate it is usually called test data. On each fold, a classifier is typically evaluated by comparing the predicted class provided by the classifier and the original class, as seen on each instance from the test set. The results are placed in the confusion matrix shown in Figure 2.2, and from this matrix, the metrics precision P and recall R are computed. After processing all the folds, the mean of precision and recall is taken, corresponding to the final result of cross-validation.

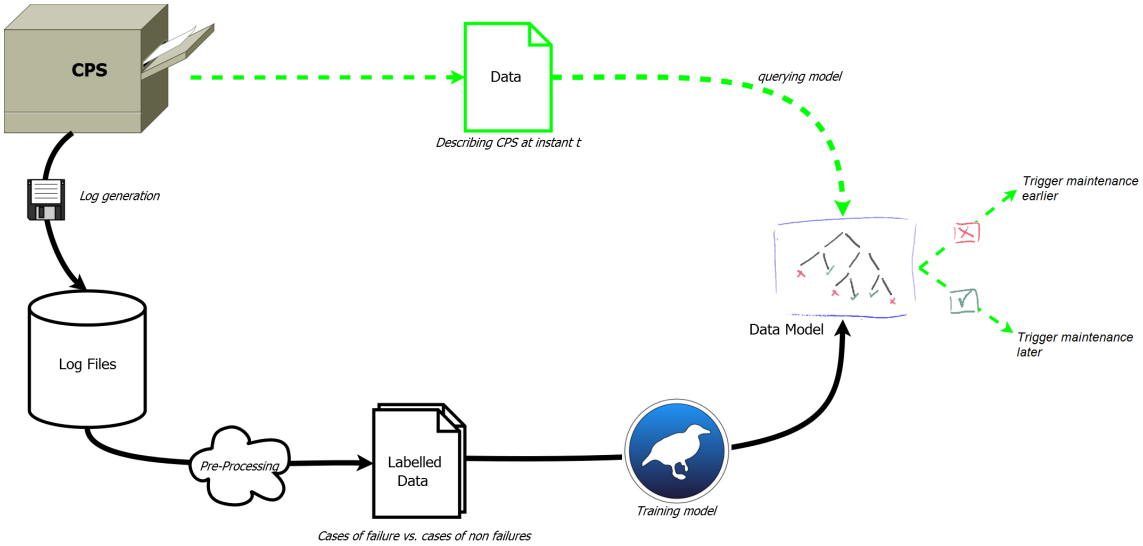


FIGURE 2.3: The process of collecting log data for a CPS component, learning classifier models, and using the outcomes as input to reschedule maintenance.

Once a classifier has been trained, it is possible to use it in order to classify new unlabeled instances. The process is as follows: in real-time, new data about the same features used to train the model are recorded and represent the state of the CPS at instant t. Such state of the system corresponds to an instance to be classified by the model. Once the outcome is provided, a decision is taken by the real-time updating method controlling the schedule of maintenance actions. The overall process of learning and using the classifier in our case, is described in Figure 2.3.

2.3.3 Related work

Extensive research has been done in the field of timed automata [5], including algorithmic and computability aspects [13, 26] and model-checking oriented research [35]. To a lesser extent, research has been developed in the context of passive learning of TA from data [142], and learning sub-classes of TA known

as event recording automata [64]. In the context of CPS, real-time online learning of TA has been investigated [98], however not related to predictive maintenance. Concerning predictive maintenance, timed automata were applied to model the operating modes in the case of the duration of tasks in manufacture scenario [133]. Statistical approaches to model the failure behaviour of CPS have already been considered [86, 108], but none of them combined the use of TA to model maintenance actions together with machine learning models.

2.4 Proposed Method and Experiments

In this section, we describe our method and the results of our experiments. Indeed, our hypothesis is that less maintenance can be done, and fewer or as many failures can occur. Hereinafter, we present the protocol observed and the data used in order to schedule nozzle-related maintenance dynamically.

2.4.1 Learning Nozzle Failure Behavior

The first step towards dynamic scheduling of nozzle maintenance actions is to build a failure behaviour model describing how the nozzles of printers are likely to fail. In order to do so, we rely on all the logs of a set of 8 printers of the year 2015. The main advantage of the logs we have at our disposal is that many metrics and nozzle-related factors are monitored continuously. Moreover, we also dealt with labelled data with data representing the nozzles at moments when they were considered as failing or not. The failing measure is the conjunction of several metrics related to print quality. We stress again the crucial role of a failure behaviour model since labelled data are costly to obtain: the model reproduces the outcome provided by printing *test pages*, a process that inevitably leads to a loss of productivity on the overall printer. We thus trained the corresponding model by selecting relevant features. To this end, we benefited from the expertise of engineers related to the field, who indicated to us the possible relevant features.

Classification Algorithm	f		!f	
	P	R	P	R
Decision Tree	0.788	0.631	0.951	0.977
Random Forest	0.626	0.579	0.943	0.953
Bayesian Network	0.683	0.809	0.973	0.949
Naive Bayes	0.329	0.424	0.918	0.882
Multilayer Perceptron	0.652	0.224	0.903	0.984

TABLE 2.1: Quality of the best classification models trained with the data.

In Table 2.1, we present the results achieved for failure detection (f). We state the results in terms of precision and recall. Precision (P) represents the proportion

of failure cases as correctly classified. Recall (R) reflects the proportion of caught failure cases among cases of failures and non-failures. We also present the results achieved for the other class (non-failure – !f). Both results are important since our scheduler uses both outcomes, i.e. to advance or postpone maintenance. The classifiers have been trained with 117k instances to classify, the same features, and evaluated with a cross-validation of 10 folds. The set was divided into 10% of instances belonging to the *failure* class and 90% to the *non-failure* class.

In our experiments, we consider the results above as good enough to be considered reliable. Moreover, we trained several classifiers (among others, decision trees, naive Bayes classifiers, neural networks, etc.), and the decision tree always performed the best. A decision tree is an interesting way of modelling the failure behaviour of a component thanks to its high understandability. As a consequence, we consider that we can safely schedule nozzle-related maintenance actions using the outcomes of the built Decision Tree.

In this case, we have used the J48 implementation of the C4.5 algorithm to learn it [118]. This algorithm builds a model from the training set using information entropy. It iteratively builds nodes choosing the attribute that best splits the current sample into subsets, using information gain. The attribute having the most significant information gain is chosen to make the decision, that is to say, to be used as the next decision node. Of course, once a subset is only composed of instances belonging to the same class, no further node is created, but a leaf instead, standing for the final class of all the instances belonging to the subset.

The fact that a decision tree can be implemented easily by a succession of *if* conditions is another reason to consider it as a premium model for industrial purposes. The lower quality of the results for the class *failure* is due to the low number of instances belonging to this class.

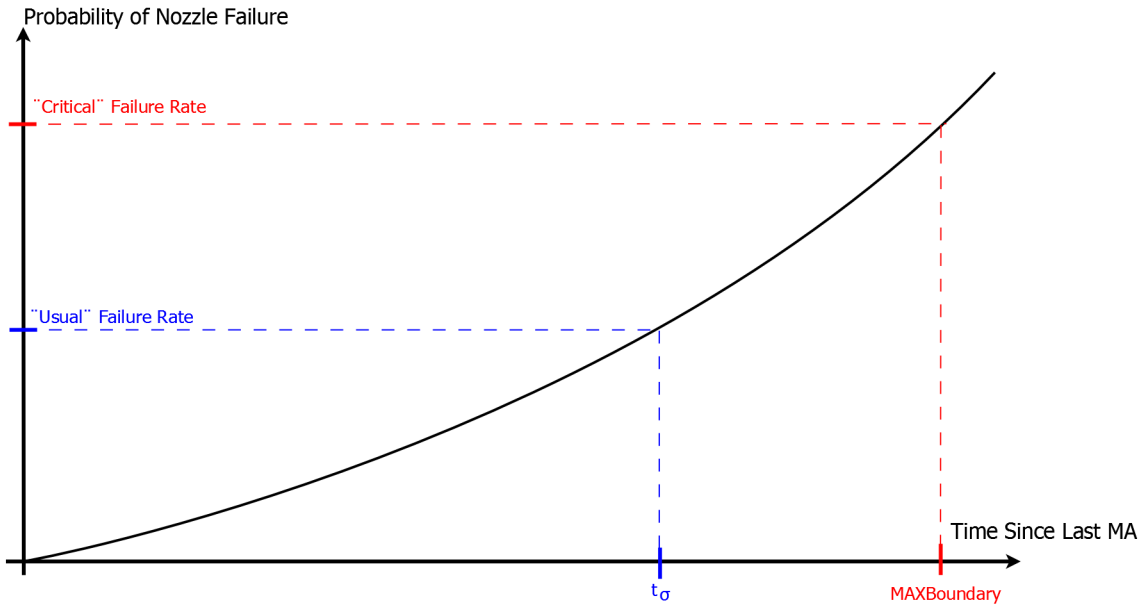


FIGURE 2.4: Nozzle failure rate in function of the time since last MA.

2.4.2 Scheduling Nozzle-Related Maintenance Actions

The idea of dynamic scheduling of maintenance actions is to rely on the outcome of a predictor (component failure behaviour model that classifies the system at instant t into two classes, failure or not failure) to put forward or backward the moment when to trigger it.

Algorithm 1 Dynamic Scheduling of a Maintenance Action.

Input:

- q : query made periodically
- y : the reducing factor for the timing of the MA
- z : the increasing factor for the timing of the MA
- t_σ : the timing where the MA is usually triggered
- $currentBoundary$: the current timing when the MA will be triggered
- $MAXBoundary$: maximum acceptable time to wait until triggering the MA

```
for each q do
    prediction ← failureBehaviorModelQuery()
    if prediction = failure                          // Advance the MA schedule
     then
        currentBoundary ← currentBoundary × y%
    end
    else
        currentBoundary ← currentBoundary × z%       // Postpone the MA
    end
    if currentBoundary is reached then
        triggerMaintenance()
        currentBoundary ← t_σ OR (currentBoundary + t_σ)/2
    end
end
```

As presented in Algorithm 1, we, first of all, consider the actual maintenance policy of the printer. Our idea is to query the classification model built previously in order to get the outcome desired. In case a failure is detected by our classification model, then the timing for all the maintenance actions triggering is decreased with a given rate of y. Otherwise, if no failure is detected in the few moments before maintenance actions are triggered, then the timed boundary to enable maintenance to occur is postponed with a given rate of z. Assuming the use of a Decision Tree as a classifier, we can notice here that the parameters y and z standing for how much to increase or decrease the maintenance clock guards can be a function of the *confidence factor* provided with each outcome performed by the classifier. This confidence factor is based on the error rate of each leaf in

the tree, hence it consists in a metric to assess how a provided outcome can be considered as reliable or not.

Finally, we distinguish two possibilities in our algorithm concerning how to reset the timed boundaries once the related maintenance has been performed. The first one consists in resetting the clocks guards to their initial values, e.g. defined in the original TA inferred from the specifications. The second one consists in resetting the future timed boundary by computing the average between the last used limit and the actual moment when the maintenance has been launched. In such a way, we assume that after several runs of the algorithm, the value computed will tend to the ideal time to wait between two maintenance actions. Both options are presented in Section 2.4.3 and the benefits between those two variants are compared.

2.4.3 Simulations using UPPAAL-SMC

In order to make simulations and evaluate the benefits of our dynamic scheduler, we used the *Uppaal-SMC* [30] model checking tool. *Uppaal* is a toolbox for verification of real-time systems represented by one or more timed automata extended with integer variables, data types and channel synchronization. The relevance of using the model checking tool *Uppaal-SMC* in our case lies in the possibility of running simulations of the system specified as a set of probabilistic timed automata.

One of the challenges of this work and its practical case study is to validate the failure behaviour model built using machine learning techniques and to use it in order to schedule maintenance actions. Of course, we had to find a way of integrating such a model in a tool like *Uppaal*, in order to benefit from the integration of the representation of the maintenance strategy (gathered from specifications of the component, and using TA), the decision tree providing insight on the failure behaviour of the nozzles (learned prior to its implementation in *Uppaal* following description in Section 2.4.1), and the function that dynamically updates the triggering of maintenance (presented in Section 2.4.2).

As shown in Figure 2.5, the whole system is composed of 3 components, all of them modelled using state-machines:

1. The *printer usage*: this models how the printer is currently being used. It is composed of 3 states, printing, standby (e.g. waiting for a print job) and sleeping (for long periods out of use). It is important to model the printer usage since the way maintenance actions are scheduled depends on the usage of the printer. Indeed, print jobs will never be interrupted to perform maintenance. It is similar when the printer is in sleeping mode since less maintenance is required when the printer is not in use.

2. The *scheduler*: this models the maintenance actions and under what conditions they are performed. In our case, we focus especially on three nozzle-related maintenance actions. Inside the specifications of the scheduler, an inner function is defined reproducing the outcome of the decision tree as

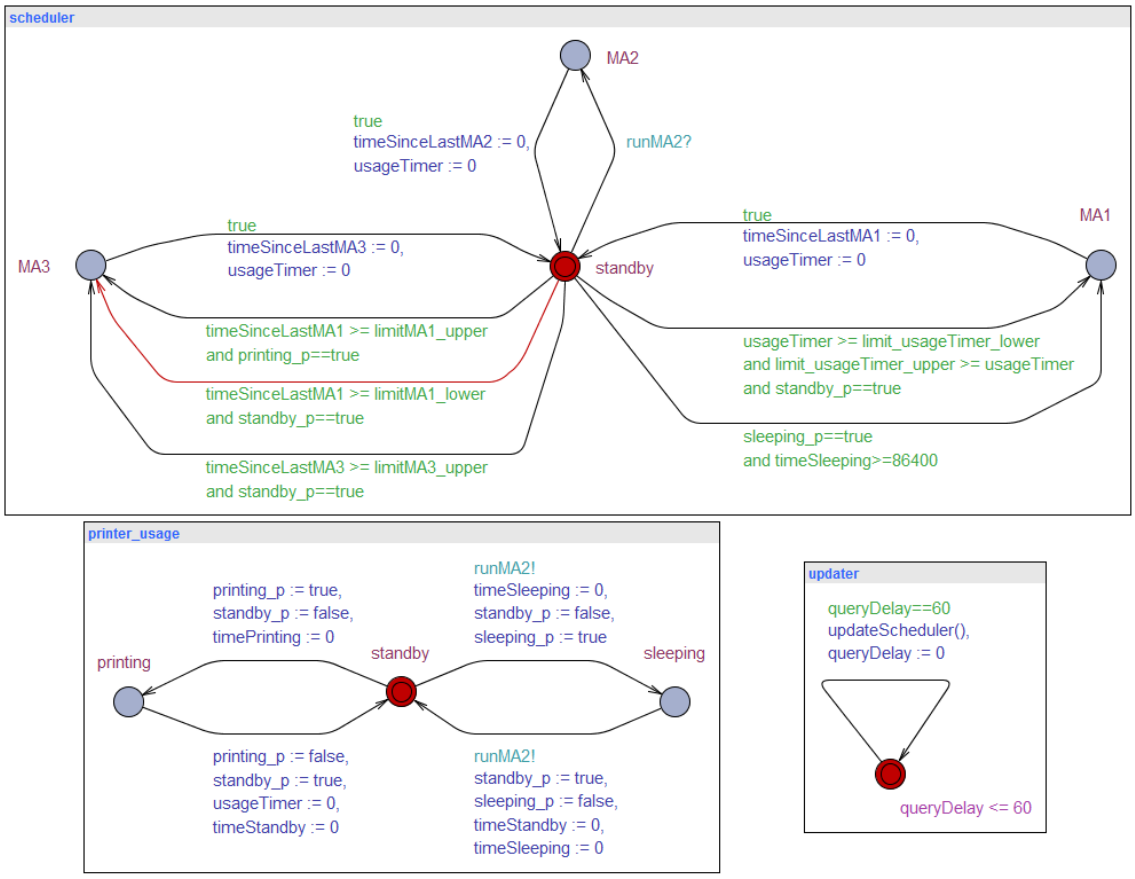


FIGURE 2.5: Designing components by state-machines in *Uppaal*.

well as the values set to the refreshed timed conditions (see Algorithm 1). The *scheduler* reflects the specifications of the nozzles under which maintenance is supposed to be performed.

3. The *updater*: this calls – with a given frequency – the function refreshing the *scheduler* from the outcome of the classification model. This function consists of a call to the decision tree implemented in *Uppaal* as an embedded function, itself consisting of a succession of *if* conditions representing the overall structure of the nozzle failure behaviour. Moreover, this function also updates the timed conditions for the triggering of maintenance actions (such as `limitMA1_upper`, `limitMA1_lower`, `limitMA3_upper`, `limit_usageTimer_lower` and `limit_usageTimer_upper` in Figure 5).

2.4.4 Results

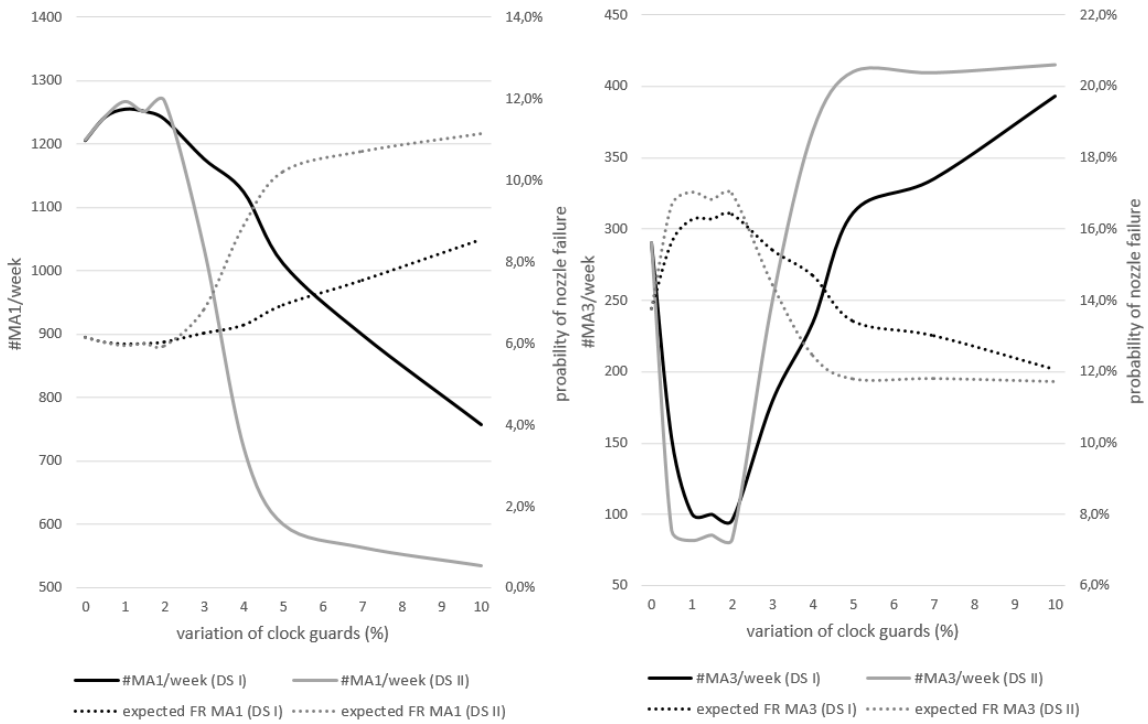
Experimental Parameters. In order to measure the benefits of our dynamic scheduler compared to a static scheduler, we relied on several metrics. First, by using our designed models in *Uppaal* and making simulations, we were able to compute how many times each state has been visited, e.g. how many maintenance actions have been triggered. That is the reason why we made 10 runs with a virtual duration of 600k seconds (approximately one week) for each configuration we wanted to test. From those runs, we were able to find out the behaviour of our scheduler in the long term, since the average of the runs is computed over 10 weeks of simulation. We were finally able to compare our results with a *static* scheduler (no call to classification model) and a *dynamic* scheduler. In the case of the dynamic scheduler, we distinguish 3 cases: the first (*Dynamic Scheduler I*) resets, once the maintenance is triggered, the values of the timed conditions to their initial values, e.g. defined in the specifications ($currentBoundary \leftarrow t_\sigma$). The second (*Dynamic Scheduler II*) aims, after a maintenance action has been performed, to average the timed conditions between the past boundary and when the MA has been done ($currentBoundary \leftarrow (currentBoundary + t_\sigma)/2$). The last one (*Dynamic Scheduler III*) computes the parameters y and z as a function of the confidence factor provided by the classification model.

In order to run experiments with *Uppaal*, several parameters have been set, such as the refreshing frequency (60 seconds), the variation of how much to postpone or advance maintenance actions (from 0.5% to 10%), the usage of the printer (the printer goes to sleeping mode every 8 hours and during at least 2 hours; after being printing during more than 2min., then, the printer can stop printing and go into standby mode; after being without printing during more than 1min., then, the printer can start printing again) and the nozzle behavior (a nozzle is likely to fail every 20 minutes). We can also mention maintenance-related additional information and settings:

- Maintenance action #1 (MA_1): according to the usage of the printer, usually triggered every 40sec – 3.5hrs. Maximum acceptable threshold set: 5hrs.

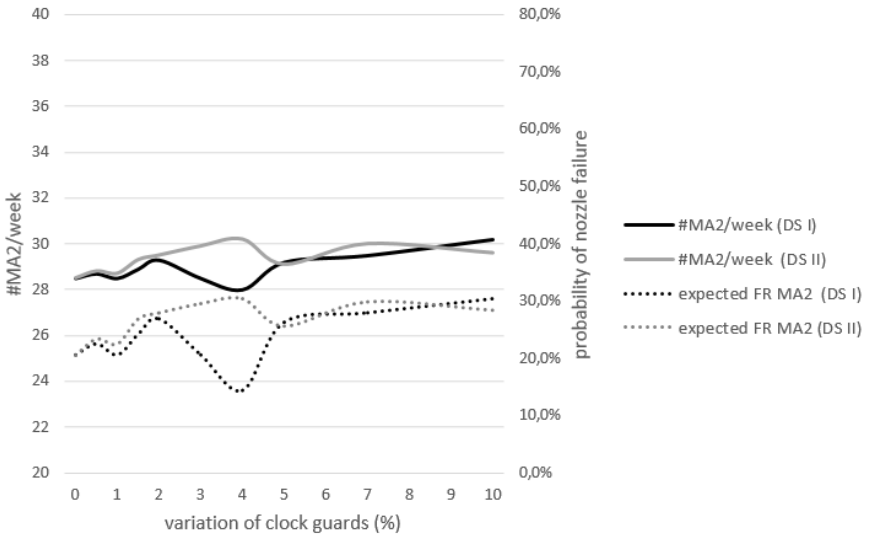
- Maintenance action #2 (MA_2): usually triggered when the system is going to and coming back from sleeping mode.

- Maintenance action #3 (MA_3): according to the usage of the printer, usually triggered every 15min. – 3.5hrs. Maximum acceptable threshold set: 5hrs.



(A) Results achieved for MA_1

(B) Results achieved for MA_3



(C) Results achieved for MA_2

FIGURE 2.6: Results concerning *Dynamic Scheduler I* and *II*. The value corresponding to a variation of clock guards of 0% stands for the *Static Scheduler*.

Results. We give a graphical representation of the number of the achieved maintenance actions performed per week as a function of the variation of the degree of clock increase/decrease, as well as the expected nozzle failure rate in Figure 2.6a to 2.6c (for *Dynamic Scheduler I* and *II*). The value corresponding to a variation of clock guards of 0% stands for the static scheduler since parameters y and z equaling 0 means no change of the clock guards. We also give the results of the

Static Scheduler and the *Dynamic Scheduler III* in Table 2.2. The results are stated for each type of MA by the number of MA performed (#MA) and the expected failure rate when the maintenance is performed (FR_{MA}). The expected failure rate is computed from the distribution shown in Figure 2.4, since the information providing the nozzle failure rate as a function of the time since last maintenance has been computed from the logs. We also present for each scheduler the proportion of cases classified as failures by our failure behavior model ($\mathcal{C}_f$ in Table 2.2 and ratio of output failures in Figure 2.7).

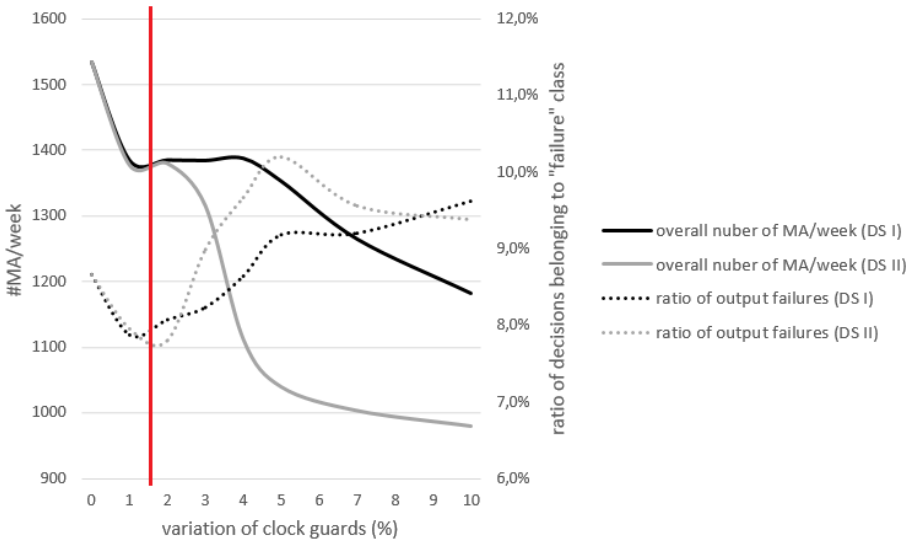


FIGURE 2.7: Overall MA performed and ratio of decisions classified as *failure*. The value corresponding to a variation of clock guards of 0% stands for the *Static Scheduler*. The vertical line shows a possible optimal variation of the clocks guards.

Discussion. From the results achieved, we can see that in some settings, the number of maintenance actions performed can decrease. Nonetheless, while the number of maintenance actions is decreasing, the *expected* nozzle failure rate slightly increases, yet still within acceptable values. We can refer to the first dynamic scheduler and to MA_3 with an optimal decrease of three times as few maintenance actions performed. The counterpart is an increase in the expected failure rate by 2 points. We can also see that in some cases, especially MA_2, our scheduler does not influence the triggering of this specific type of maintenance. Indeed,

Scheduler	#MA_1	FR_{MA_1}	#MA_2	FR_{MA_2}	#MA_3	FR_{MA_3}	$\mathcal{C}_f$
Static	1205.6	6.1%	28.5	20.5%	290.3	13.8%	8.7%
Dynamic III	806.6	8.2%	29.1	25.6%	378.1	12.3%	9.5%

TABLE 2.2: Number of maintenance actions triggered using *Dynanic Scheduler III*.

MA_2 is an example of usage-based maintenance, whereas our method only modifies timed-based maintenance. According to the results achieved by the *Dynamic Scheduler III* e.g. advancing or postponing the clock guards using the confidence factor of the Decision Tree, we can see that less MA_1 is performed but the expected failure rate when MA_1 is triggered is increased by 2 points, whereas MA_3 is done more often. Furthermore, according to the number of cases detected as failure cases by our classifier as well as the number of actions performed (whatever the type), we can see that in some cases, less maintenance is done, and fewer failures are detected by our model, in particular when using a variation of how much to postpone or advance maintenance actions of 1 or 2%. This is shown in Figure 2.7 by a red line. This result proves our assumption stating that we can at the same time save maintenance and improve the print quality. Finally, for some settings and independently of the scheduler, our dynamic schedulers perform more actions than a static scheduler.

2.5 Conclusion

In this chapter, we describe a new method for dynamic maintenance scheduling. We expect that our method can be generalized to other domains. The major novelty we bring, to the best of our knowledge, is a method involving machine learning techniques and model checking. It uses the decision of a classifier in order to put forward or postpone when maintenance should be triggered, i.e. how to update scheduling of automatic periodic maintenance defined by a TA, and simulates its benefits using the model checking tool *Uppaal*.

According to our results, we can conclude that our dynamic scheduler reduces the number of actions performed in different settings. We also note that the price to pay in order to do less maintenance is a slight increase in the expected failure rate. Thus, depending on how critical the component is, our technique can reduce maintenance costs for a negligible increase in the risk of breakdown. In our case, it entailed an insignificant loss of print quality. Moreover, in some settings, we could reduce the failure rate as well as the number of maintenance performed. We also believe that our technique can be particularly interesting in the case of the unavailability of sensors that provide direct information about component failures. The strength of an embedded decision model is its availability at any time. Furthermore, we expect that applied to other real systems, our technique could achieve similar results to those found in our simulation.

Concerning further work, we think that within the scope of our case study, we can extend the current experiment not only to nozzles but also to other related components, or at least components that share related maintenance actions. Furthermore, in future, we will look into the use of Fault Trees [126]. Indeed, we believe that a fault tree pattern can be used to model interactions between several components and how failures can propagate from one component to others. This matter is raised in Chapters 3 and 4. We also hope that extending such a technique to the use of real-time automata can enhance the schedulability and control of CPS. We can also enhance the scheduler by taking into account the timing

occurrences of anomalies [97]. Finally, we could orientate the choice of the machine learning techniques used towards stream mining tools and algorithms [18], which would additionally offer the possibility of updating the failure behaviour model the more new labelled data are available. Then, it could be possible to deal with unseen events or combination of parameters, and keep an accurate model throughout the life of the CPS.

Chapter 3

Fault Trees from Data: Efficient Learning with an Evolutionary Algorithm

Cyber-physical systems come with increasingly complex architectures and failure modes, which complicates the task of obtaining accurate system reliability models. At the same time, with the emergence of the (industrial) Internet-of-Things, systems are more and more often being monitored via advanced sensor systems. These sensors produce large amounts of data about the components' failure behaviour, and can, therefore, be fruitfully exploited to automatically learn reliability models. This chapter presents an effective algorithm for learning a prominent class of reliability models, namely fault trees, from observational data. Our algorithm is evolutionary in nature; i.e., is an iterative, population-based, randomized search method among fault-tree structures that are increasingly more consistent with the observational data. We have evaluated our method on a large number of case studies, both on synthetic data, and industrial data. Our experiments show that our algorithm outperforms other methods and provides near-optimal results.

3.1 Introduction

Reliability engineering is an important field that provides methods, tools and techniques to evaluate and mitigate the risks related to complex systems such as drones, self-driving cars, production plants, etc. Fault tree analysis is one of the most prominent technique in this field. It is widely deployed in the automotive, aerospace and nuclear industry, by companies and institutions like NASA, Ford, Honeywell, Siemens, the FAA, and many others. Further, fault tree analysis is standardized by the IEC [42].

Fault trees (FTs) are graphical models that represent how component failures arise and propagate through the system, leading to system-level failures. Component failures are modelled in the leaves of the tree as *basic events*. Fault tree *gates* model how combinations of basic events lead to a system failure, represented by the top event in the FT. The analysis of such FTs [127] is multifold: they can be used to compute dependability metrics such as system reliability and availability;

understand how systems can fail; identify the best ways to reduce the risk of a system failure, etc. A key bottleneck in fault tree analysis is, however, the effort needed to construct a faithful fault tree model. FTs are usually built manually by domain experts. Given the complexity of today's systems, industrial FTs often contain thousands of gates. Hence, their construction is a very intricate task, and also error-prone, since their soundness and completeness largely depends on domain expertise.

With the emergence of the industrial Internet-of-Things [9], Cyber-physical systems are more and more equipped with smart sensor systems, monitoring whether or not a system component is in a failed state or not. Even though such a monitoring system is often designed to detect failures during operations, their data can be very fruitfully deployed to learn reliability models. Such data can be crucial for the engineers to build an FT [73]. Recent work focused on learning FTs from observational data, identifying causalities from data [105]. In this chapter, we focus on FT generation from data, using an evolutionary algorithm (EA). Evolutionary algorithms approximate stochastic learning by mimicking biological evolution, and have been successfully applied to a broad plethora of applications; examples include the scheduling of flexible manufacturing systems [62], automata learning [52], induction of Boolean functions [114], and many more.

In our case, each stage of the EA keeps a population of candidate FTs. New fault trees are generated by mimicking biological evolution, that is by reproduction (e.g., adding or deleting FT gates), crossover (e.g. swapping FT branches), and mutation (e.g. changing an AND gate into an OR gate). In total, we have identified seven (parametric) generation rules, which are equally applied. Finally, we select the new population by only keeping those FTs with the best *fitness*, i.e. with the best fit to the observational data.

We have experimentally verified the applicability of our algorithm, both on synthetic data, an industrial case study, and a benchmark of FTs previously studied in the literature. Our experiments show that the algorithm is fast and accurate. Further, we have investigated which of those rules lead to successful mutants and found that all rules are needed to obtain optimal solutions.

Being a first step, our algorithm focuses on *static* fault trees, featuring only Boolean gates. An important topic for future work is the extension to dynamic fault trees. These come with additional gates, catering for common dependability patterns like spare management and functional dependencies. Static fault trees, however, have appeal as relatively simple yet powerful formalism, and are often used in practice. Furthermore, dynamic fault trees strongly depend on the temporal order in which failures occur and their learning will, therefore, require more complex data, such as time series.

This chapter is organized as follows. Section 3.2 reviews related work on learning FTs from data. In Section 3.3 we recall preliminary definitions. We present then in Section 3.4 our technique to infer an FT using an EA. In Section 3.5 a further variation of our EA, in order to take expert knowledge into account. In Section 3.6 we show the results we achieved. Finally, we discuss and conclude about further research.

3.2 Related Work

Related work on learning fault trees spans three areas of research: the synthesis of fault trees from other graphical models of the system under study; recent work on the generation of fault trees from observational data describing the system; and, since fault trees are in essence Boolean functions, literature on learning Boolean functions from observational data.

Model-based synthesis. While state-of-the-art fault tree design is still performed manually by domain specialists [75], several methods have been proposed to synthesize FTs automatically from other models of the system [20, 28, 132]. Thus, these methods require the pre-construction of a system model in a suitable model description language, which varies with each method for FT synthesis.

For example, the HiP-HOPS framework [113] synthesizes an FT from a system model describing transactions among the system components, annotated with failure information. Similar synthesis methods were developed from the AltaRica system description language, which models the causal relations between system variables and events using transitions [88]. Certain system control models in the form of directed graphs have also been shown to suit the synthesis of fault trees [4, 69], as well as Go models [150].

Furthermore, system models described in the model language NuSMV also enable the synthesis of FTs; a limitation of this is the fact that the resulting FTs show the relation between top events and basic events, but do not show how failure propagates in the system via system components [27]. Static or dynamic FTs fault trees can also be synthesized from models in the Architectural Analysis and Design Language AADL [89].

As a special case, FT generation has been attempted so that the learning method includes explicit reasoning about the causal relations between events in the system. For this type of FT generation, [85] requires a probabilistic system model, from which a model-checking step obtains a set of probabilistic counterexamples. When the system is concurrent, the order of events in these counterexamples does not necessarily signify causality, so logical combinations of events are separately validated for causality. Similarly, in [90] a cause-effect graph (and from that, an FT) is extracted by model checking a process already modelled by a finite-state machine, against safety and liveness requirements, using failure injection.

Since model-based FT learning require prior modelling of the system under study, these methods do not *adapt* well in applications where the systems evolve and thus need to be remodelled, e.g., components are replaced or the interactions between components change, thus changing the failure modes and their probability of occurrence.

Learning causal models from data. Supervised *automated learning* of dependability models using *system data*, unlike the model-based methods described above, will adapt to system change, under the assumption that all the system components remain monitored by sensors throughout their lifetime, also after a change

of components. Here, we take "learning" to mean broadly any autonomous computational intelligence method able to infer (even approximate) high-level models of knowledge from data.

Causal Bayesian Networks [87] are standard graphical models which been learnt from data examples. These models have straightforward translations into FTs, but are themselves NP-hard or require exponential time to synthesize accurately [40, 77]. These networks will model a limited form of causality, namely global causal relationships, rather than a sequence of causal relationships among events local to the components of a system.

LIFT [105] is a recent approach for learning static FTs with Boolean event variables, n-ary AND/OR gates, annotated with event failure probabilities. The input to the algorithm is untimed observational data, i.e., a dataset where each row is a single observation over the entire system, and each column records the value of a system event. All intermediate events to be included in the FT must be present in the dataset, but not all may be needed in the FT, and a small amount of noise in the dataset can be tolerated. LIFT also includes a causal validation step (the Mantel-Haenszel statistical test) to filter for the most likely causal relationships among system events, but the worst-case complexity is exponential in the number of system events in the data. Its main advantage is that of being one of the few automated FT-learning methods which specifically validate causality.

Learning Boolean formulas and classifiers from data. Before LIFT, observational data was used to generate FTs with the IFT algorithm [96] based on standard decision-tree statistical learning. The advantage of learning a graphical decision tree out of data is the inherent interpretability of decision-tree models, and their ease of translation into other graphical models.

Boolean formulas or networks were also machine-learnt using a similar tree-based method [110, 77]. The classic C4.5 learning algorithm [118] yields a Boolean decision tree easily translatable into a Boolean formula by constructing the conjunction of all paths leading to a leaf modelling a True value (i.e., system failure), and then simplifying the Boolean function. The resulting models encode the same information as a decision tree (i.e., a classifier for the observational data), so lack the validation of causal relations, but are expected to preserve their predictive power about the system. This retained our attention: indeed, static FTs (in opposition to dynamic FTs, where time-dependence of events is considered) can be seen as Boolean functions. Furthermore, Boolean formulas were also machine-learnt using black-box classifiers (namely, classifiers not easily interpretable as a graphical model). Such methods include Support-Vector Machines [129], Logistic Regression [16], and Naive Bayes [128].

We propose a novel algorithm to learn an FT that best (most accurately) classifies records in a tabular dataset composed of observational tuples, in which values (failures) for each Boolean basic event and Boolean top event in the system are known. We compare it with a subset of these existing learning algorithm, in terms of its performance when fitting data.

3.3 Background

In this section, we first define the structure of a static FT (consisting of logic gates, and also of intermediate events). Essentially, *intermediate events* in the FT are logical combinations of other intermediate events, with only *basic events* as the leaves of the tree, and one special intermediate event called the *top event* as root.

Then, we also formally define a dataset from which an FT is then inferred. Intuitively, the dataset is a collection of records, each of which is a valuation of a set of Boolean variables: whenever a variable is True, a *failure* was observed for that system component. Every such variable models the state of one basic, indivisible system component and may thus appear as a basic event in the FT. The *top event* of the FT must be a variable in the dataset and can be seen as the outcome to predict by the FT. The formulations below follow definitions from [105].

3.3.1 Fault Trees

FTs [143] are trees that model how component failures propagate to system failures. Since subtrees can be shared, FTs are in fact directed acyclic graphs (DAGs) rather than trees. The leaves of the tree (or rather sinks of the DAG) model component failures and are called *basic events (BEs)*.

Gates model how BE failures lead to system failures. Standard fault trees feature two types of gates: AND, and OR. An AND gate fails if all its children fail; an OR gate fails if at least one its children fails.

Definition 3.3.1. *A **gate** G is a tuple $(t, \boldsymbol{I}, O)$ such that:*

- *t is the type of G with $t \in \{And, Or\}$.*
- *$\boldsymbol{I}$ is a set of $n \geq 2$ intermediate events $\{i_1, \ldots, i_n\}$ that are inputs of G.*
- *O is the intermediate event that is the output of G.*

We denote by $I(G)$ the set of intermediate events in the input of G and by $O(G)$ the intermediate event in the output of G.

Definition 3.3.2. *An **AND** gate is a gate $(And, \boldsymbol{I}, O)$ where output O occurs (i.e. O is True) if and only if every $i \in \boldsymbol{I}$ occurs.*

Definition 3.3.3. *An **OR** gate is a gate $(Or, \boldsymbol{I}, O)$ where output O occurs (i.e. O is True) if and only if at least one $i \in \boldsymbol{I}$ occurs.*

Definition 2 requires that all system components modelled by the events in the input of the **AND** gate must fail in order for the system modelled by the event in the output to fail. Similarly, Definition 3 requires that one of the system components modelled by the events in the input of the **OR** gate must fail in order for the system modelled by the event in the output to fail.

Definition 3.3.4. *A **basic event** B is an event with no input and one intermediate event as output. We denote by $O(B)$ the intermediate event in the output of B.*

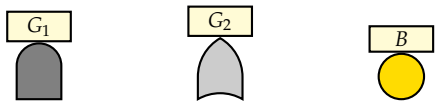


FIGURE 3.1: Graphical notation for **AND** (G_1) and **OR** (G_2) gates, and for a basic event (B).

Sometimes other gates are considered, like the XOR (exclusive OR), the voting gate and the NOT gate [143, 83]. For the sake of simplicity, we focus on the AND and OR gates; other gates can be treated in a similar fashion. The root of the tree is called the *top event* (T). The *top event* represents the failure condition of interest, such as the stranding of a train, or the unplanned unavailability of a satellite. Thus, an FT fails if its *top event* fails.

Definition 3.3.5. *A **fault tree F** is a tuple $(\boldsymbol{BE}, \boldsymbol{IE}, T, \boldsymbol{G})$ where:*

- ***BE** is the set of basic events; $\forall B \in \boldsymbol{BE}$, $O(B) \in \boldsymbol{IE}$. A basic event may be annotated with a probability of occurrence p.*
- ***IE** is the set of intermediate events.*
- *T is the top event, $T \in \boldsymbol{IE}$.*
- ***G** is the set of gates; $\forall G \in \boldsymbol{G}, I(G) \subset \boldsymbol{IE} \cup \boldsymbol{BE}, O(G) \in \boldsymbol{IE}$.*
- *The graph formed by **G** should be connected and acyclic, with the top event T as unique root.*

*We denote by $IE(\boldsymbol{F})$ the set of intermediate events in **F** and by $IE(G)$ the intermediate event corresponding to gate G.*

Fig. 3.2 shows an FT modeling a lamp failure. The top OR-gate shows that a lamp fails, if there is either a button failure, or a battery failure. A button failure happens if either an operator (**OF**) or a cable (**CF**) fails. The AND-gate indicates that a battery failure happens if both batteries are low (**LB I** and **LB II**).

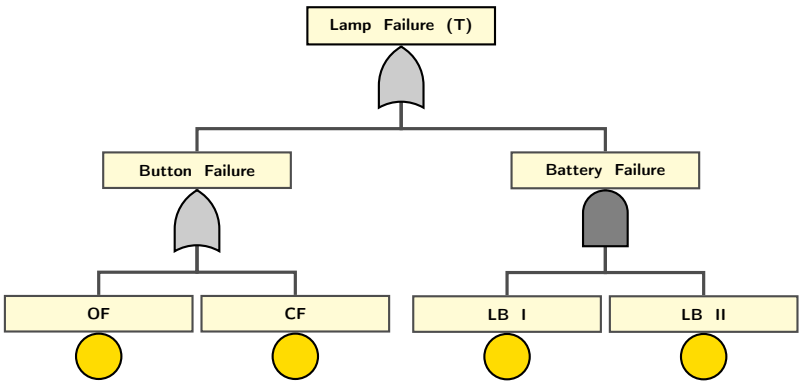


FIGURE 3.2: Example of a Fault Tree.

3.3.2 Dataset for Fault Trees

We define a dataset from which we learn a fault tree as a collection of records. Each record is a valuation for the set of BEs, indicating whether a *failure* was observed for that BE. Further, we assume that our dataset is *labeled*, i.e., also indicates whether the top event T has failed, yielding the predicted outcome of the FT.

Definition 3.3.6. *A record R over the set of variables $\boldsymbol{V}$ is a list of length $|\boldsymbol{V}|$ containing tuples $[(V_i, v_i)], 1 \leq i \leq |\boldsymbol{V}|$ where V_i is a variable name, $V_i \in V$ and v_i is a Boolean value of V_i.*

Definition 3.3.7. *A dataset $\boldsymbol{D}$ is a set of r records, all over the same set of variables $\boldsymbol{V}$. Each variable name in $\boldsymbol{V}$ forms a column in $\boldsymbol{D}$, and each record forms a row. When k identical records are present in $\boldsymbol{D}$, a single such record is shown, with a new count column for the value k.*

Table 3.1 shows an example dataset for the FT from Fig. 3.2.

OF	**CF**	**LB I**	**LB II**	**T**	count
0	0	0	0	0	900
0	0	0	1	0	15
0	0	1	0	0	5
0	0	1	1	1	25
0	1	0	1	1	5
0	1	1	0	1	5
1	0	0	0	1	35
1	0	1	0	1	5
1	1	0	0	1	3
1	1	1	0	1	2

TABLE 3.1: Example dataset for FT in Fig. 3.2.

3.4 Learning Fault Trees with Nature-Inspired Stochastic Optimization

Evolutionary algorithms (EAs) were among the earliest artificial intelligence methods, first envisioned by Alan Turing in 1950 [140]. EAs are heuristics that mimic biological evolution: one starts with an initial population, and iteratively generates new individuals through modification and recombination, where only the best individuals are kept in the next generation – mimicking survival of the fittest. EAs are particularly suitable to automatically learn models of some kind, such as trees, graphs or matrix structures, free-form equations, sets and permutations, and synthetic computer programs, etc. Evolutionary algorithms have been very successfully applied in a number of domains, ranging from antenna designs for

spacecrafts [71], graph-like network topologies [29], matrix-like robot designs [39], and the internal structure of artificial neural networks [84].

The main challenges in devising EAs are (a) formalizing what is a syntactically correct *solution* to the problem (in our case, a well-formed fault tree), and (b) formalizing what makes, semantically, a solution better than another, i.e., writing a *fitness function* which takes any proposed solution and returns a numerical "goodness" for that solution. In our case, we want the fault tree to be consistent with the observational data, so the fitness function is the proportion of records correctly classified; the EA will aim to *maximize* this as close as possible to its optimal value of 100%.

Then, the EA consists of an iterative optimization process, which maintains a *population* of candidate solutions at each iteration, and randomly mutates and combines (i.e., applies *genetic operators* to) solutions from a population, such that the fitness of the *best solution* per population improves in time. The evolutionary framework is shown in Fig. 3.3.

A. *Initialization*: The initial population contains two simple FTs; all variables in the input dataset $\mathbf{D}$ are represented as BEs in these FTs.

B. *Mutation and recombination*: Genetic operations are performed on the FTs with a fixed probability, and generate new FTs.

C. *Evaluation*: The fitness of each new FT is evaluated.

D. *Selection*: High-fitness FTs from the new generation replace low-fitness FTs from the previous generation.

E. *Termination*: Steps B to D are repeated until a given termination criterion is met. This can be if at least one solution in the population exceeds a given fitness bound, or if a given maximum number of iterations is reached. Upon termination, the best solution or solutions in the population are returned.

We describe these steps in more detail below.

3.4.1 Initialization

Our evolutionary algorithm takes as input a dataset $\mathbf{D}$ and aims at computing a fault tree with maximal fitness to this dataset. The dataset yields the set of BEs, as well as the top level event T.

We start with an initial population consisting of the following two FTs, where all BEs are connected to T via an AND and an OR gate respectively:

- $F_1 = (\mathbf{BE}, \{T\}, T, \{(And, \mathbf{BE}, T)\})$
- $F_2 = (\mathbf{BE}, \{T\}, T, \{(Or, \mathbf{BE}, T)\})$

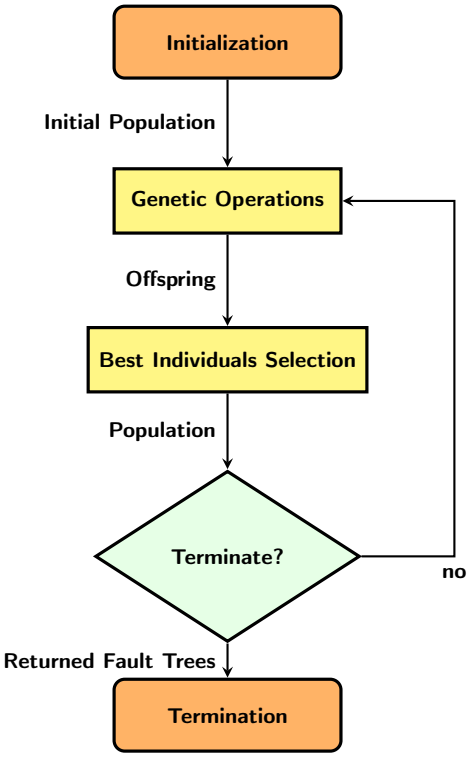


FIGURE 3.3: Evolutionary learning framework.

where $\mathbf{BE} \cup \{T\} = \mathbf{V}$, so all the basic events and the top event must be in the dataset $\mathbf{D}$. These two FTs are the simplest structures including all the observational variables in the data, with an AND gate and an OR gate, respectively, at the top of the FT, and all BEs as inputs of this gate. These two individuals act as a seed population; later populations are larger in size. The population size is a setting depending on the nature of the FT to learn (namely, the number of BEs). Since the time complexity of the algorithm depends on the population size, increasing the population size may lead to scalability issues. It has also been shown in [38] that increasing the population size does not always perform as well as expected. Thus, in our experiments we limit the population to hundreds of FTs.

3.4.2 Mutation and recombination

We define seven stochastic genetic operators (one binary and six unary) which apply to FTs. For each iteration, each of the genetic operators operate on all individuals in the population with a given probability, to create new individuals. The order in which they are applied is randomized at each iteration.

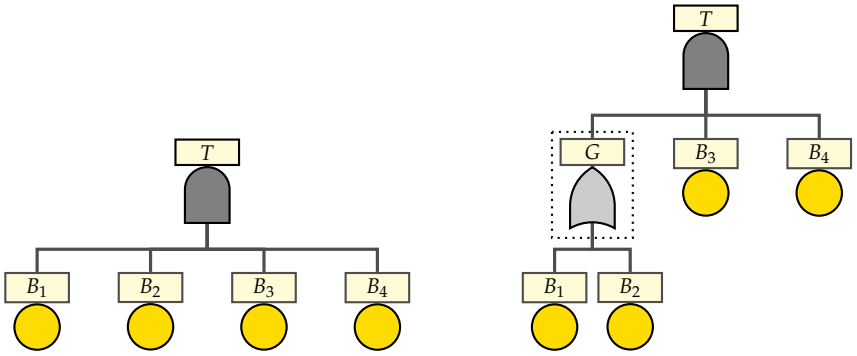


FIGURE 3.4: Example of *G-create*.

G-create. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$, create a gate $G \notin \mathbf{G}$, randomly select its nature (AND or OR), then randomly select a gate $G' \in \mathbf{G}$. Randomly select inputs events I' of $I(G')$ to become inputs of G such that $I(G) = I'$ and $I(G') = I(G') \setminus I'$. Then, add $O(G)$ to the input events of G' such that $I(G') = I(G') \cup O(G)$. The new FT is $\mathbf{F} = (\mathbf{BE}, \mathbf{IE} \cup O(G), T, \mathbf{G} \cup \{G\})$ with $G \notin \mathbf{G}$. This operator is illustrated in Fig. 3.4.

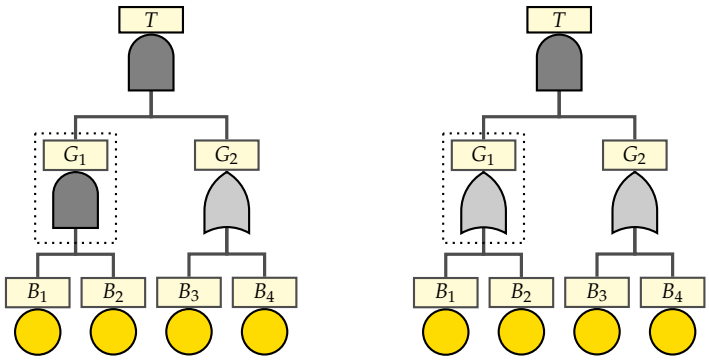


FIGURE 3.5: Example of *G-mutate*: mutating gate G_1 into an OR gate.

G-mutate. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$, randomly select a gate $G \in \mathbf{G}$ and change its nature (AND to OR, or OR to AND). This operator is illustrated in Fig. 3.5.

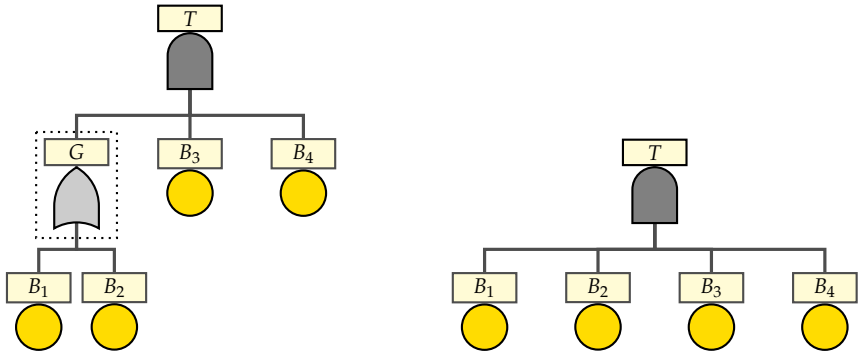


FIGURE 3.6: Example of *G-delete*: deleting gate G.

G-delete. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$, randomly select a gate $G \in \mathbf{G}$ such that $O(G) \neq T$ and delete it. We set then, $I(G_p) = \bigcup_{i \in I(G)} IE_i$ such that $O(G) \in G_p$. The new FT is $\mathbf{F} = (\mathbf{BE}, \mathbf{IE} \setminus O(G), T, \mathbf{G} \setminus \{G\})$. This operator is illustrated in Fig. 3.6.

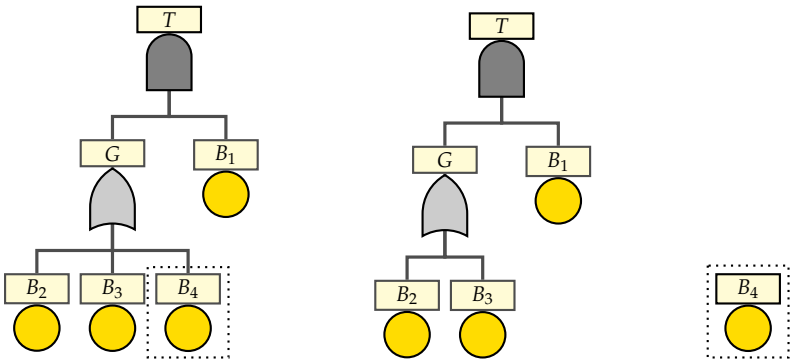


FIGURE 3.7: Disconnecting basic event B_4 from an FT.

BE-disconnect. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$ and a randomly chosen basic event $B \in \mathbf{BE}$, disconnect B from its intermediate event $G = O(B)$. The new FT if $\mathbf{F} = (\mathbf{BE} \setminus \{B\}, \mathbf{IE}, T, \mathbf{G})$, where $B \notin I(G)$.

Note here that a gate G left with 0 or 1 input will not be removed: this is to preserve the solution search space, and enable the connection of BEs to this gate. Only the genetic operator *G-delete* can remove such a gate. This operator is illustrated in Fig. 3.7.

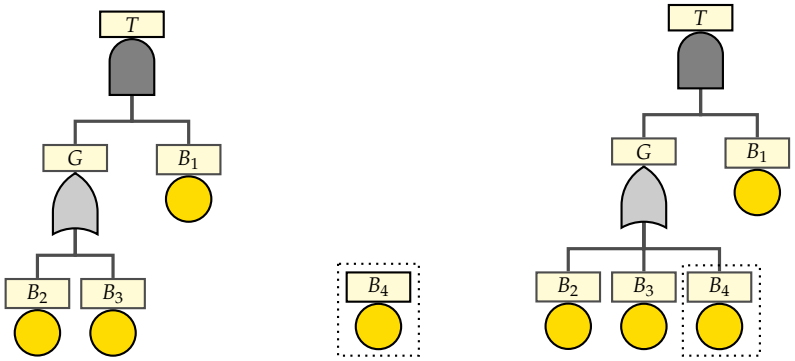


FIGURE 3.8: Example of *BE-connect*: connecting B_4 to an FT.

BE-connect. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$ and a basic event $B \notin \mathbf{BE}$ and $B \in \mathbf{V} \setminus T$, randomly choose a gate $G \in \mathbf{G}$ and connect B to the input of G. The new FT is $\mathbf{F} = (\mathbf{BE} \cup \{B\}, \mathbf{IE}, T, \mathbf{G})$, where $B \in I(G)$. This operator is illustrated in Fig. 3.8. Note that this operator is essentially the inverse of **BE-disconnect**.

The relevance of this operator lies in the fact that some FTs in the population may not contain as many BEs as variables $\mathbf{V}$ in the dataset. Also, our definition of this operator implies that no BE will be input to 2 different gates. However, the connection of the same BE to 2 different gates can occur within a *crossover* operation.

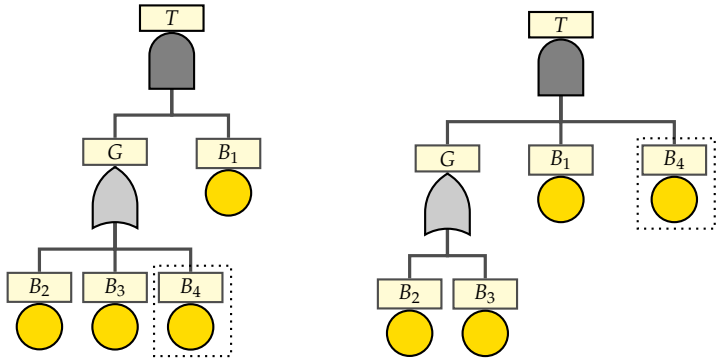


FIGURE 3.9: Example of *BE-swap*: swapping basic event B_4 from gate G to gate T.

BE-swap. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$ and a randomly chosen basic event $B \in \mathbf{BE}$ and a randomly chosen gate $G \in \mathbf{G} \setminus O(B)$, disconnect B from $O(B)$ and connect B to G. This operator is illustrated in Fig. 3.9.

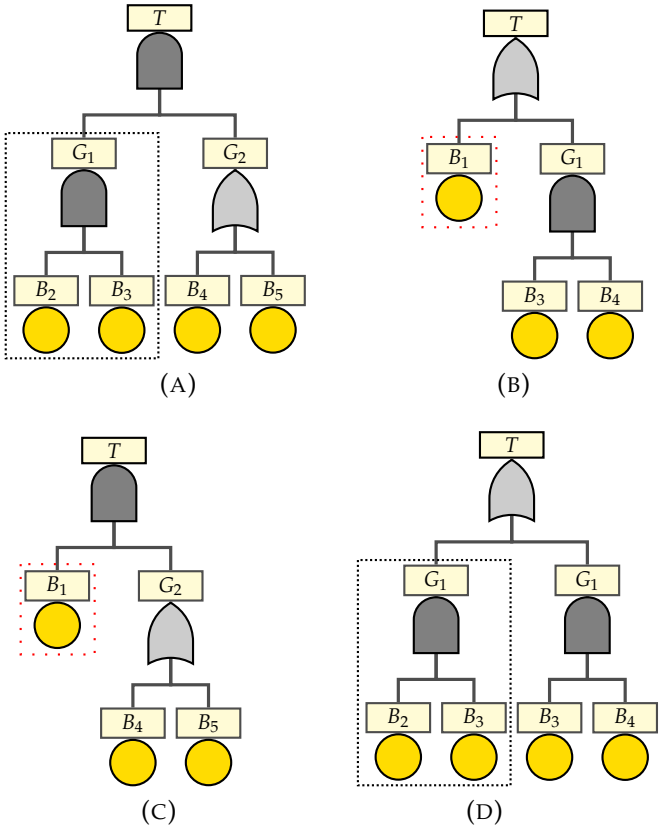


FIGURE 3.10: Example of *crossover* between (A) and (B) and resulting children (C and D).

Crossover. The crossover operator takes two FTs as input and swaps at random two of their subtrees. This leads to two new FTs, where the first fault tree contains the selected subtree of the second fault tree and vice versa. More precisely, one selects at random an intermediate event $IE_1 \in \mathbf{IE_1}$ from the first fault tree $\mathbf{F_1}$, and one also selects at random an intermediate event $IE_2 \in \mathbf{IE}_2$ from the section fault tree. Then one replaces in $\mathbf{F_1}$ the subtree under $\mathbf{IE_1}$ by the subtree under $\mathbf{IE_2}$. Similarly, one replaces in $\mathbf{F_2}$ the subtree under $\mathbf{IE_2}$ by the subtree under $\mathbf{IE_1}$.

Given two input FTs $\mathbf{F_1} = (\mathbf{BE_1}, \mathbf{IE_1}, T_1, \mathbf{G_1})$ and $\mathbf{F_2} = (\mathbf{BE_2}, \mathbf{IE_2}, T_2, \mathbf{G_2})$, randomly select an $IE_1 \in \mathbf{IE_1}$ and $IE_2 \in \mathbf{IE_2}$. Then, we set $I(O(IE_1)) = I(O(IE_1)) \setminus IE_1 \cup IE_2$ and $I(O(IE_2)) = I(O(IE_2)) \setminus IE_2 \cup IE_1$. Finally, $O(IE_1) = O(IE_2)$ and vice versa.

This operator is shown in Fig. 3.10. Note that in this example, resulting child in Fig. 3.10c contains a BE (BE_3) connected to multiple gates.

3.4.3 Evaluation

We define the *fitness* of an FT as the number of records in the dataset for which the value of the top event, given the values of the BEs, is correctly computed.

Definition 3.4.1. *The fitness of a fault tree is its accuracy w.r.t. the dataset* $\mathbf{D}$ *such that*

$$f = \frac{\sum_{r \in D} x}{\sum_{r \in D} k} \text{ where } \begin{cases} x = k & \text{if } V[T] = P[T] \\ x = 0 & \text{otherwise} \end{cases}$$

where $P[T]$ *stands for the predicted value of the top event given the dataset* $\mathbf{D}$ *for a given FT,* $V[T]$ *the real value of the top event and* k *the number of occurrences of the record* r*.*

3.4.4 Selection

The selection strategy of best individuals to undergo genetic operations is important to increase the improvement rate of the fitness of the population. Among others, FTs can be selected according to the following strategies:

- *Elitism*: systematically selects the best fitted individuals. This guarantees that the solution quality obtained by the EA will not decrease from one generation to the next.
- *Roulette wheel Selection*: also known as *fitness proportionate selection*, individuals in the population will be selected proportional to their fitnesses.
- *Tournament Selection*: involves running several "tournaments" among a few individuals chosen at random from the population. The winner of each tournament (i.e. the individual with the best fitness) is selected.
- *Stochastic Universal Sampling*: chooses several individuals from the population by repeated random sampling, and uses a single random value to sample all of the individuals by choosing them at evenly spaced intervals. This allows least fitted individuals to be selected.
- *Random Selection*: the FTs are randomly selected in the population.

In all our experiments, we use an elitist strategy. This choice is motivated by the nature of the individuals: best fitted FTs are the closest in the population to the optimal solution. They consist of Boolean gates and BEs, which means that a least fitted solution needs more genetic operations to become optimal. We thus hope that mutating the best FTs will be less costly in terms of iterations of the EA in order to converge towards a good solution. In Section 3.6.3, we conducted an experiment justifying the relevance of elitism to our case, where we make a comparison between the different selection strategies mentioned above.

3.4.5 Termination

The decision whether to return the best FTs in the actual population or to continue the evolutionary process follows the termination criteria. In our experiments, the termination criteria where:

1. at least one solution in the population achieves an accuracy of 1, which means a perfect fitness to the data.
2. a maximum number of allowed iterations is reached.
3. convergence: no improvement of the best FT in the population has been observed for a given number of iterations.

Note that several runs of our EA may return different FTs with the same fitness, especially in terms of structure. Indeed, two FTs with a different structure may be semantically equivalent. We show in Fig. 3.20 how an equivalent FT to a target FT is learnt by our EA. Indeed, FTs, alike Boolean formulas, can be factorized to Disjunctive Normal Form (DNF, where a Boolean formula is standardized as a disjunction of conjunctive clauses) or to Conjunctive Normal Form (CNF, where a Boolean formula is standardized as a conjunction of disjunctive clauses). As a result, it may happen that all variables in the dataset do not appear in the FT, since they are either not needed or not relevant. In our experiments, we chose to compute the CNF of the best-fitted FT. The transformation of an FT into a CNF is based on the following rules: the double negative law, De Morgan's laws, and the distributive law.

3.5 Learning of Partial Fault Trees

A fruitful application of learning fault trees is the learning of partial models, where domain experts partially know the structure of the FT, and other parts need to be inferred from data. In this way expert knowledge and data driven approaches are aggregated.

To accommodate this approach, we propose here a variation of our EA. We parameterize our EA with such a partially known structure as input to the algorithm. The task for the EA is then to evolve *sub*-Fault Trees, given the known *skeleton* of the FT. The initial population becomes then the partial structures given as input. Genetic operators are slightly modified to ensure the given skeletons to remain unmodified in each mutated FT. We gather at any moment of the evolutionary process a population composed of FTs containing the allowed skeleton.

3.5.1 Initialization

In the same way as the procedure described in Sect. 3.4, our evolutionary algorithm takes as input a dataset $\mathbf{D}$, as well a known FT-*skeleton* F_o. We start then with an initial population consisting of this FT-*skeleton*:

$$F_o = (\mathbf{BE_o}, \mathbf{IE_o}, T_o, \mathbf{G_o})$$

where $\mathbf{BE_o}$ are BEs contained in the skeleton, $\mathbf{IE_o}$ are the intermediate events of the skeleton, T_o is the top event of the skeleton and $\mathbf{G_o}$ is the set of gates contained in the skeleton.

This is this structure given as input that will remain in all mutated FTs all along the evolutionary process.

3.5.2 Mutation and recombination

In order to preserve the FT-*skeleton* during mutation and recombination operations, we have to adapt the following genetic operators.

G-create-o. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$, create a gate $G \notin \mathbf{G}$, randomly select its nature (AND or OR), then randomly select a gate $G' \in \mathbf{G}$. Randomly select inputs events I' of $I(G') \setminus \mathbf{IE_o}$ to become inputs of G such that $I(G) = I'$ and $I(G') = I(G') \setminus I'$. Then, add $O(G)$ to the input events of G' such that $I(G') = I(G') \cup O(G)$. The new FT is $\mathbf{F} = (\mathbf{BE}, \mathbf{IE} \cup O(G), T, \mathbf{G} \cup \{G\})$ with $G \notin \mathbf{G}$. In that way, we allow the creation of a gate such that its input events are not in the gates of the skeleton, that is, the skeleton remains unchanged.

G-mutate-o. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$, randomly select a gate $G \in \mathbf{G} \setminus \mathbf{G_o}$ and change its nature (AND to OR, or OR to AND).

G-delete-o. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$, randomly select a gate $G \in \mathbf{G} \setminus \mathbf{G_o}$ such that $O(G) \neq T$ and delete it. We set then, $I(G_p) = \bigcup_{i \in I(G)} IE_i$ such that $O(G) \in G_p$.

BE-disconnect-o. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$ and a randomly chosen basic event $B \in \mathbf{BE} \setminus \mathbf{BE_o}$, disconnect B from its intermediate event $G = O(B)$. The new FT if $\mathbf{F} = (\mathbf{BE} \setminus \{B\}, \mathbf{IE}, T, \mathbf{G})$, where $B \notin I(G)$.

BE-swap-o. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$ and a randomly chosen basic event $B \in \mathbf{BE} \setminus \mathbf{BE_o}$ and a randomly chosen gate $G \in \mathbf{G} \setminus O(B)$, disconnect B from $O(B)$ and connect B to G. This operator is illustrated in Fig. 3.9.

Crossover-o. Given two input FTs $\mathbf{F_1} = (\mathbf{BE_1}, \mathbf{IE_1}, T_1, \mathbf{G_1})$ and $\mathbf{F_2} = (\mathbf{BE_2}, \mathbf{IE_2}, T_2, \mathbf{G_2})$, randomly select an $IE_1 \in \mathbf{IE_1} \setminus \mathbf{IE_o}$ and $IE_2 \in \mathbf{IE_2} \setminus \mathbf{IE_o}$. Then, we set $I(O(IE_1)) = I(O(IE_1)) \setminus IE_1 \cup IE_2$ and $I(O(IE_2)) = I(O(IE_2)) \setminus IE_2 \cup IE_1$. Finally, $O(IE_1) = O(IE_2)$ and vice versa.

Note that the scenario where expert guidance is used to lead to a faster convergence of the FT learning algorithm is realistic: indeed, this variation is helpful to refine existing FTs, or checking if a handmade model is accurate given real-world measurements.

3.6 Experimental Evaluation

We have evaluated the efficiency and effectiveness of our EA method using a large number of cases. We compared our methods with six other learning techniques: five approaches from the literature, and the variant of our own EA technique for learning partial fault trees. For these methods, we investigated both the accuracy and as well as runtime. Our comparisons were performed for a set of synthetic cases (Sections 3.6.2 and 3.6.3), as well as for industrial benchmarks (Section 3.6.4 and 3.6.5).

3.6.1 Experimental set up

The first three methods in our evaluation are: (1) Support Vector Machine (abbreviated svm in the figures) [129], (2) Logistic Regression (abbreviated log) [16] and (3) Naive Bayes Classifier (nba) [128]. These methods are Boolean classifiers that, given the values of the BEs, predict the value of the top event T. During our experiments, we used state-of-the-art Python implementations for these techniques [116]. Being classifiers, methods (1) - (3) do not yield fault tree models, only a prediction for the value of T.

Then, we have used three methods that do learn fault tree models: (4) We have compared our results to the well-known C4.5 algorithm for learning decision trees [118] (abbreviated c45). Decision trees can be transformed to FTs, by first computing in the decision tree the conjunction of all paths leading to failure leaves, and then simplifying the conjunction to CNF. (5) We have also compared to the earlier LIFT [105] approach, which re returns an FT. (6) Finally, we used the variation of the EA for learning partial fault trees (ea-p). Here, we assumed that the two upper layers of the fault trees were fixed.

To compare these methods, the observational data was divided into 2 sets: one training set, used as input to the EA (with an average of 2/3 of all possible observations), and a test set containing all observational variables (complete boolean table), used to evaluate the solution returned by our algorithms. The parameters of the EA were set as follows: we used a population size of 100. As termination criteria, we used either a maximum number of 100 iterations or an observed convergence (i.e. no improvement of the best individual's fitness) over 10 iterations, or an FT with fitness 1 (optimal solution) in the population. Each genetic operator, was applied with probability 0.9 in order to increase the mutation rate of the

population. The selection and replacement strategy were elitists, to systematically replace least-fitted individuals in the old population by the best individuals in the union of the old population and the set of newly generated individuals. Finally, to homogenize several runs of the EA, the conjunctive normal form (CNF) of the best FT was returned in the termination step.

3.6.2 Synthetic Dataset: accuracy and runtime

We have first used a large synthetic case. We considered 100 randomly generated fault trees with 6 to 15 BEs, and for each FT a randomly generated data set, with 200 to 230k records. Figures 3.11 and 3.12 present respectively the average accuracy and the average runtime, both as functions of the number of BEs.

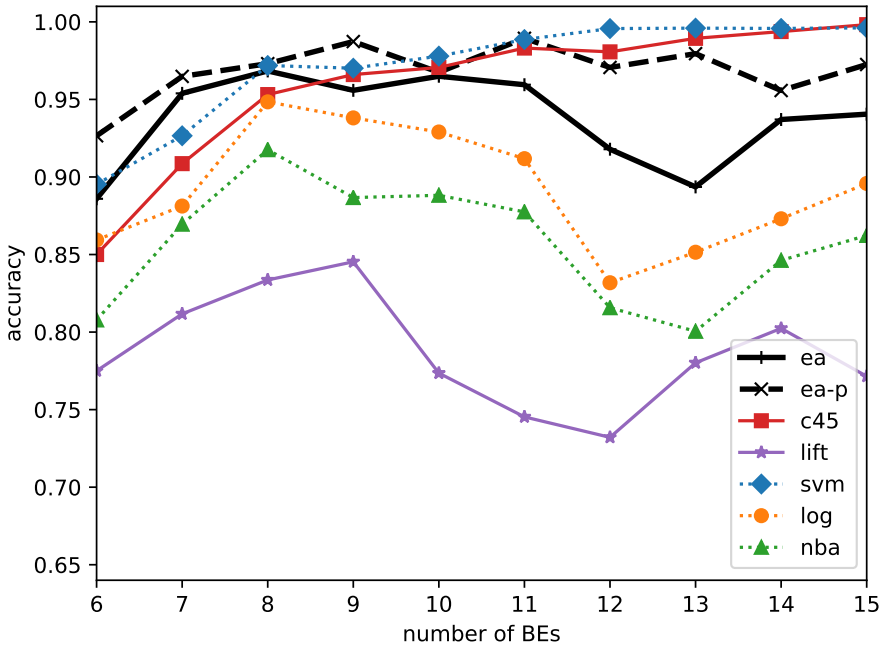


FIGURE 3.11: Accuracy as a function of the number of BEs.

Fig. 3.11 shows that the svm and c45 methods have the highest accuracy. However, the svm method only provides a classifier, not a fault tree. Further, the c45 method does not perform well in terms of runtime, see Fig 3.12. Our methods EA and EA-p perform reasonably well in terms of accuracy, as well as in term of run time. Finally, the log and nba method are fast, but provide low accuracy. We can see that, despite a good accuracy, first learning a decision tree (C4.5), then the simplification of the conjunction of all paths leading to failure leaves has scalability issues. Indeed, the computation of the CNF of a Boolean expression is exponential regarding the number of clauses it contains. Hence the observed blow up in the context of C4.5, especially in case of overfitting (which can lead to a multiplication of the paths leading to failure leaves). Conversely, our methods show a

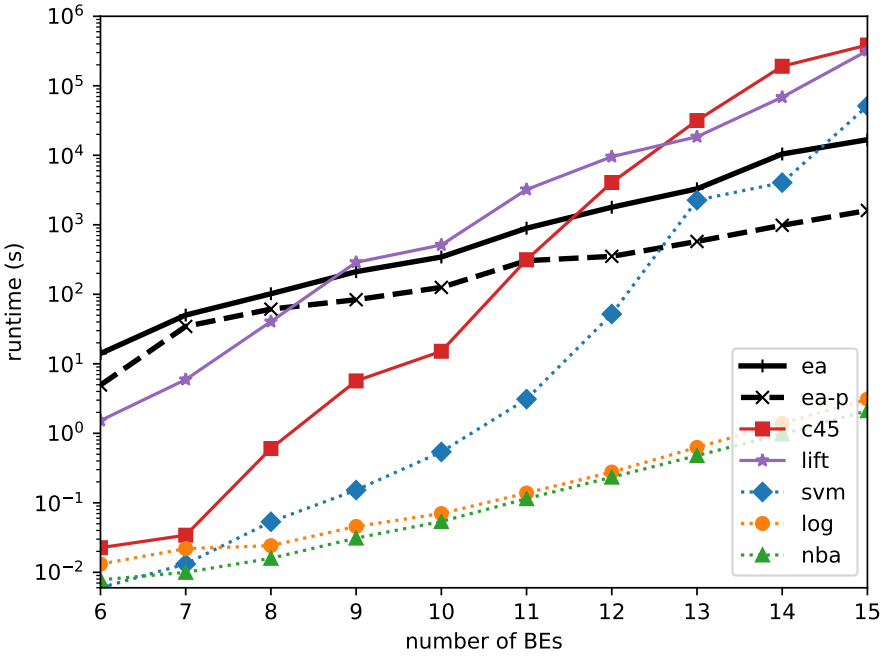


FIGURE 3.12: Runtime as a function of the number of BEs.

better scalability, especially for systems the FT of which contains a large number of BEs. Even if the obtained accuracy is lower that C4.5, it is still reasonable even in an industrial context. Finally, we see that LIFT obtains less good results. This can be explained by an exponential complexity, and the fact LIFT requires data about intermediate events.

We can also see that the more the FTs contain BEs, the more it is complicated to gather a solution with perfect fitness w.r.t. the training set. This is due to the great number of iterations needed to converge to an optimal solution when dealing with a large number of BEs. However, we can see that expert knowledge is extremely beneficial in the case of ea-p, where the skeleton of the FT is given. It enables to learn more accurate FTs (accuracy $> 95\%$) and faster (up to 10 times faster than the baseline EA).

3.6.3 Synthetic Dataset: other statistics

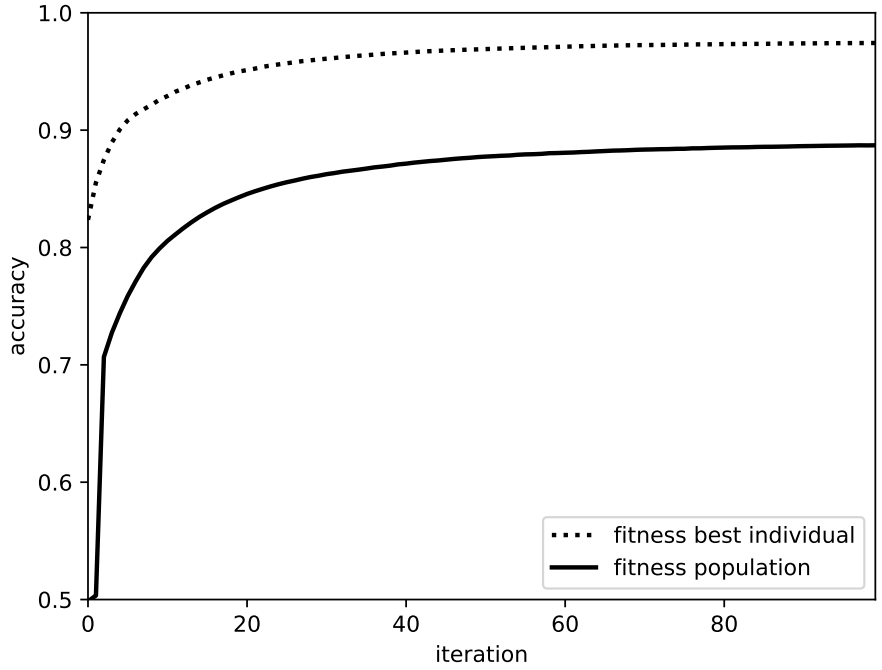


FIGURE 3.13: Average FT accuracy during the EA.

FT accuracy through iterations We also explored the behaviour of the EA method over time. In particular, we investigated, as a function of the number of iterations, the average fitness of the best element in the population and the average fitness of the whole population. This is shown in Fig. 3.13.

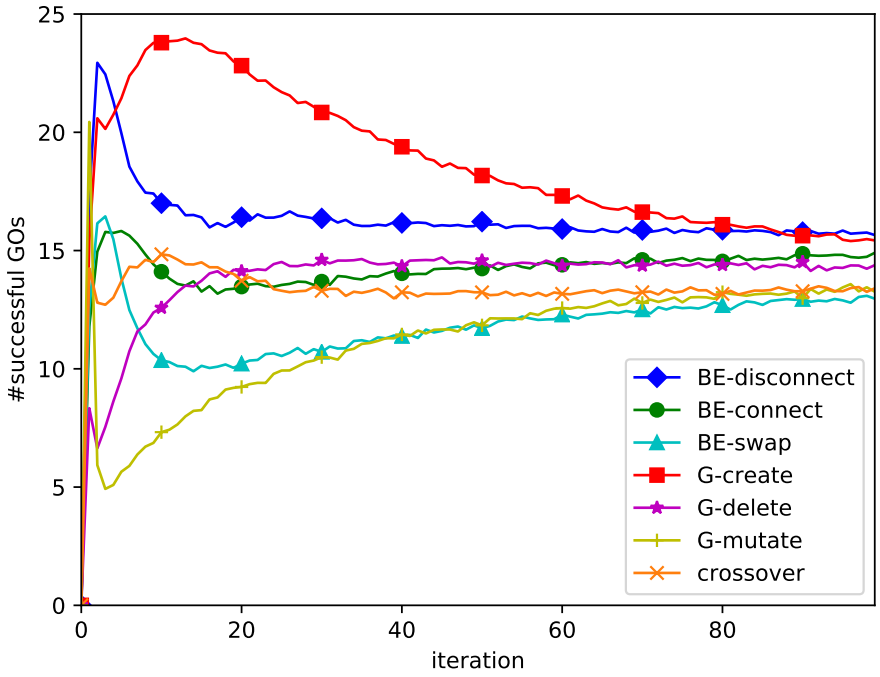
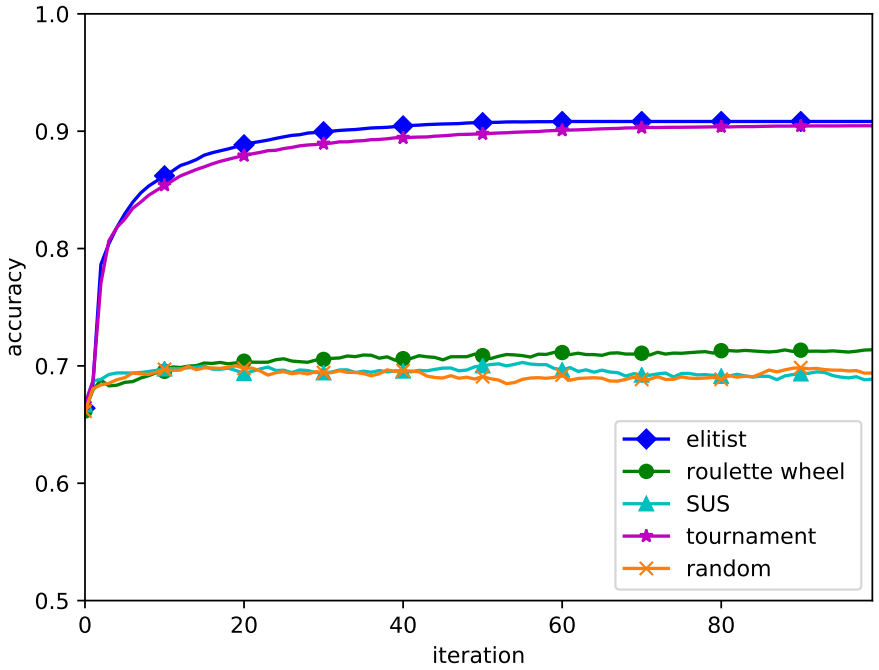
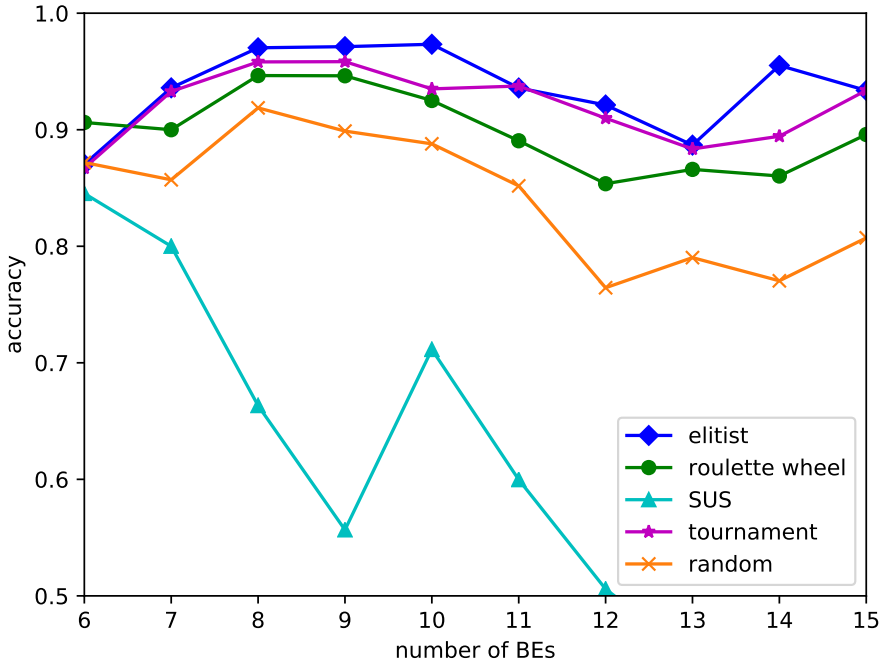


FIGURE 3.14: Average number of successful Genetic Operations (GOs) per type.

Successful GOs Further, we also investigated which genetic operations were successful, as a function of the number of iterations, shown in Fig. 3.14). The latter is computed by looking at, for each iteration, the number of individuals issued from the same genetic operator who survived in the next generation, i.e. whose fitness was good enough to be kept in the population. We see that the success of most operations depends on the stage of the EA: this is the case for *BE-disconnect*, and *G-create*, which provide satisfying new individuals during the first iterations of the algorithm. An explanation is that *G-create* will increase the size of the FT, i.e. its complexity. Then, the search space of the solution is increased. In opposition to these gates, *G-mutate* seems to be a less good operator, since the number of individuals issued from it tends not to survive in the population. This is mainly due to the change of semantics this operator implies: indeed, when the depth of the mutated gate is small (i.e. close to the top event), the meaning of the resulting FT may drastically change. Hence a small number of successful operations of this type.



(A) Average FT accuracy during the EA.



(B) Average FT accuracy.

FIGURE 3.15: Comparison of different selection strategies.

Selection strategies In order to justify the elistist selection strategy, we compare it to roulette wheel, stochastic universal sampling, tournament and random selections. The results of this experiment are shown in Fig. 3.15. We show in Fig. 3.15a how the accuracy of the best FT in the population evolves through iterations. In Fig. 3.15b, we show the average FT accuracy obtained as a function of the number of BEs in the FT. These results show that the elitist strategy leads to the highest FT accuracy, and to the fastest convergence.

3.6.4 Case Study with Industrial Dataset

We present here an industrial case study based on the dataset from [92]. We consider here a component called the nozzle. The system containing the nozzles records large amounts of data about the state of the components over time, among them the failing of nozzles and nozzle-related factors.

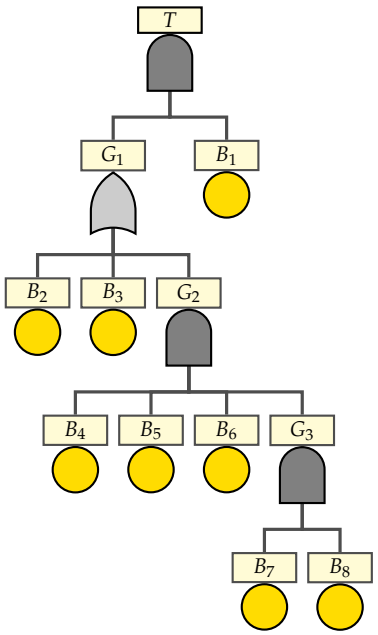


FIGURE 3.16: Fault Tree for the nozzle case study learned by our evolutionary algorithm.

The dataset is composed of 9,000 records, 8 basic events being nozzles-related factors and a Boolean top event, standing for nozzle failure. We ran our genetic algorithm 10 times, with a maximum allowed iterations of 100 (convergence criterion of 10), and the fitness of the best FT learnt was of 0.997 (split ratio for train/test set of 80/20). The resulting fault tree is shown in Figure 3.16).

Even in a practical context, multiple runs of the EA may return different (possibly equivalent) FTs, with different structures. Depending on the applications, expert knowledge can figure out whether one or the other returned FT is the most relevant to the case study. To help the selection process, one can place additional constraints on the fault tree, such as the number of children.

3.6.5 Fault Tree Benchmark

We present here the results we obtained for a set of publicly available benchmark suite[1], consisting of industrial fault trees from the literature:

- Cardiac Assist System (CAS) is a model of a hypothetical cardiac assist system with redundant CPUs, motors, and pumps [25].
- Container Seal Design Example (CSD) models a sealing mechanism for a container that is to return samples to Earth [135].
- Multiprocessor Computing System (MCS) models a computer system consisting of three power supplies, three memory modules and two groups of two hard disks [102].
- Monopropellant Propulsion System (MPS) models a monopropellant propulsion system which provides thrust for the small space flight vehicle while in orbit [135].
- Pressure Tank (PT) is a model of a pressure tank-pump-motor device and its associated control system [135].
- Sensor Filter Network (SF14) is a synthetic example which contains a number of sensors that are connected to some filters [74].
- Spread Mooring System (SMS_A1) is a model of a spread mooring system under extreme environmental conditions, for different tanker-buoy configurations of anchors and buoys available in the Marmara Sea near Yarimca [100].

The structural composition of these Fault Trees (stated in terms of number of gates, number of BEs, probability of the top event and probabilities of BEs) is shown in Table 3.2.

	# gates	# BEs	T prob	BEs probs
CAS	8	9	3e-6	[1e-6, 5e-6]
CSD	4	6	1e-6	[1e-5, 1e-1]
MCS	10	11	6e-6	[2e-6, 8e-2]
MPS	9	9	2e-5	[5e-5, 2e-2]
PT	5	6	4e-5	[5e-6, 1e-4]
SF14	8	9	3e-3	[2e-3, 3e-3]
SMS_A1	7	10	3e-2	[6e-8, 9e-3]

TABLE 3.2: Statistics on datasets.

Whereas the fault tree models were given in the literature, no data sets were available. Therefore, we have randomly generated these data sets, containing

[1] https://dftbenchmarks.utwente.nl/

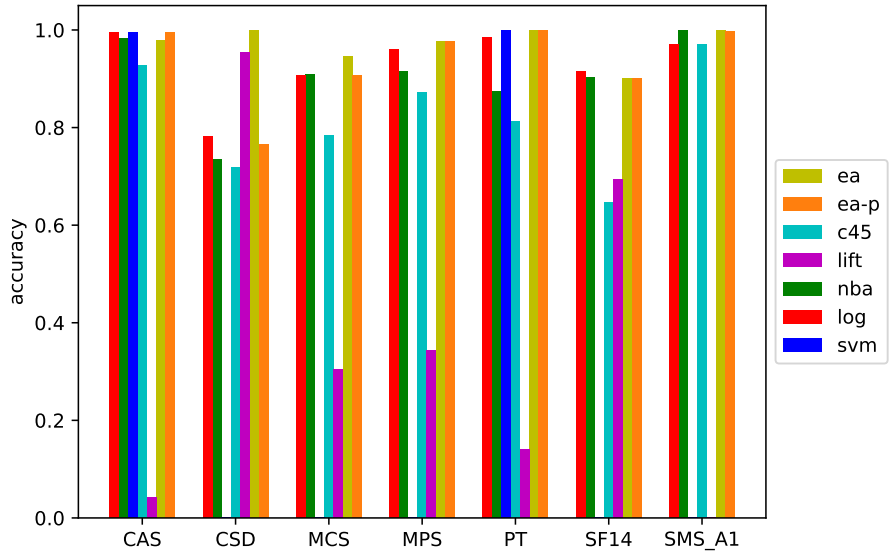


FIGURE 3.17: Results of Fault Tree Benchmark (accuracy).

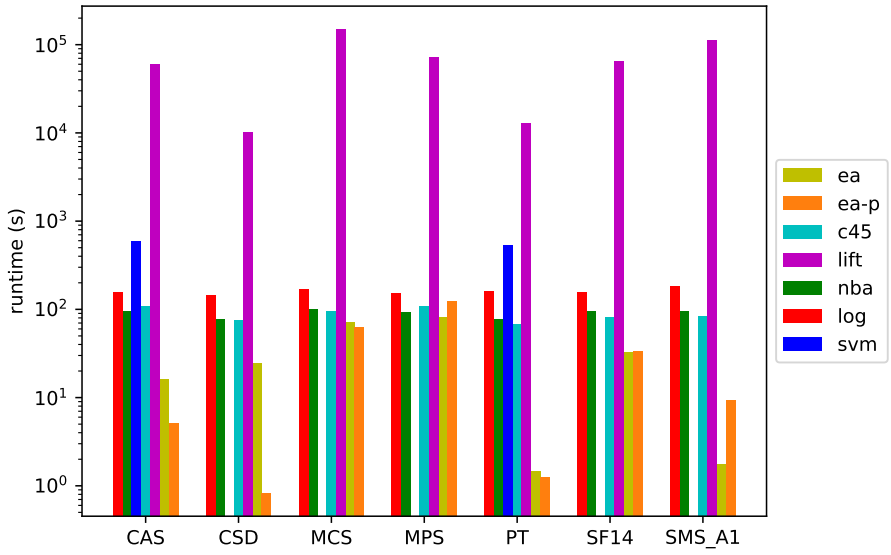


FIGURE 3.18: Results of Fault Tree Benchmark (runtime).

10M records per case. Since the benchmark does provide failure probabilities per BEs, we have used those probabilities: If p_e is the failure probability of BE e in the benchmark, then we set, in each data record, $R[e] = 1$ with probability p_e.

Fig. 3.17 and 3.18 present the accuracy and runtime, respectively. Missing bars stand for experiments for which no result could have been obtained within 1 week of running time.

We can see that for all case studies, our method is either the most, or the second most, efficient. We also see that in all cases, our method is among the most accurate methods.

Note that our implementations and dataset are available[2].

3.7 Discussion

Extensions. The definition of gates and genetic operators can be extended. We show how to deal with **K/N** gates, which are gates of type $(k/N, \mathbf{I}, O)$ where output O occurs (i.e. O is True) if at least k input events $i \in \mathbf{I}$ occurs, with $|\mathbf{I}| = N$. The cardinality of a k/N gate is said to be the number k.

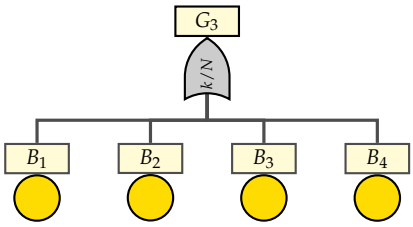


FIGURE 3.19: A k/N gate with 4 input events.

Note that this gate can be replaced by the OR of all sets of k inputs, but the use of k/N gates is much more compact for the representation of an FT. We can then define new genetic operators, such as **k-n-change** where, given an input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$, randomly select a k/N gate $G \in \mathbf{G}$ and change its cardinality, such that $k \in [1, N-1]$.

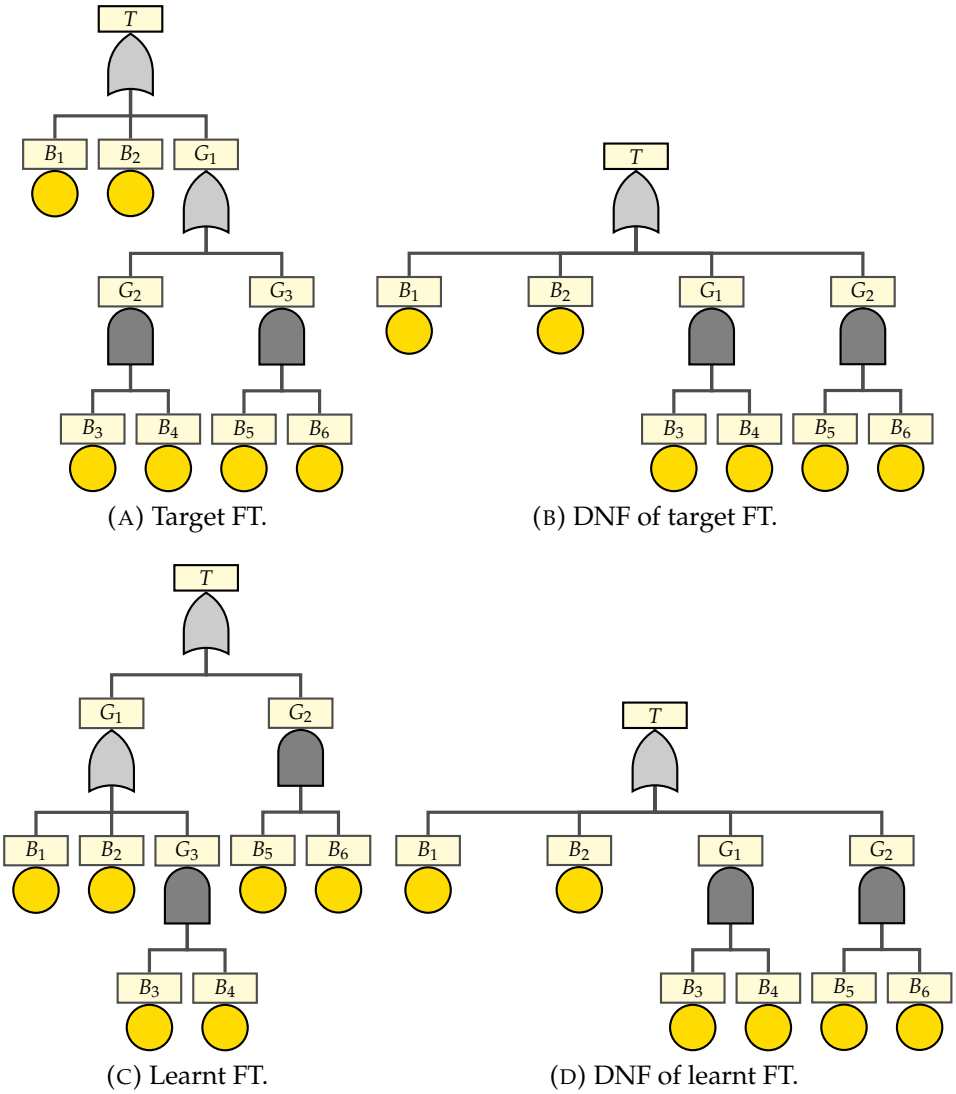
We also extend the mutate gate operator of **G-mutate**, as follows: Given an input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$, randomly select a gate $G \in \mathbf{G}$ and change its nature (AND to OR or k/N ; OR to AND or k/N ; k/N to OR or AND). Similarly, we can redefine the create gate operator **C-create** such that the randomly selected nature of the new gate is chosen among AND, OR or k/N.

Limitations. While we can accurately learn small fault trees, the main limitation of our method at the moment is scalability. While other techniques, especially naive Bayesian classifiers, score well, techniques that learn models experience slower performance. Therefore, a solution may be to combine both methods. We can also use better heuristics on which GO to deploy, and with what parameters. Such ideas were also the key to the success of EAs in other application domains.

The result obtained by the EA does not ensure a perfect fitness of the FT with regards to the data. This is the case when a maximal number of iteration has been reached, and the best FT in the population returned. Hence the near-optimality of our algorithm. In addition, multiple runs of the EA may return different (possibly equivalent) FTs, with different structures. We leave for further work the discovery of which of the returned FT is right, based on the data, figuring out causal relationships between variables using Mantel-Haenszel Partial Association score [19].

[2] https://gitlab.science.ru.nl/alinard/learning-ft

Another limitation lies in the growing size of the FTs after iterations: this may lead to overgrown FTs. However, we think that the best fit individuals may be compacted when returned: indeed, some gates in the FTs may contain none of only one input, and some factorization can be applied. We thus recommend performing FT reduction on the returned FTs, such as the calculation of CNFs or DNFs. Alternatively, the size of the solutions can be taken into account as a second fitness function for the selection step. The implementation of such a multi-objective EA [47] is left for further work.



(A) Target FT. (B) DNF of target FT.

(C) Learnt FT. (D) DNF of learnt FT.

FIGURE 3.20: Fault Trees may have different equivalent forms.

3.8 Conclusion and Future Work

We presented an evolutionary algorithm for automated generation of FTs from Boolean observational data. We defined a set of genetic operators specific to the formalism of FTs. Our results show the robustness and scalability of our algorithm. However, for FTs with basic events greater than 10, the approximated FT learned by our algorithm is still of sufficient quality ($> 99\%$) for industrial purposes. We also applied our technique to an industrial case study and a benchmark of FTs to ensure its relevance.

Our future research will focus on the learning of dynamic FTs, and especially trying to learn their specific gates such as PAND, FDEP and SPARE gates. We will also further look into Bayesian Inference and translating rules from Bayesian Networks to FTs. This is discussed in Chapter 4.

Chapter 4

Learning Fault Trees through Bayesian Networks

Cyber-physical systems have increasingly intricate architectures and failure modes, which is due to an explosion of their complexity, size, and failure criticality. While expert knowledge of individual components exists, their interaction is complex. For these reasons, obtaining accurate system reliability models is a hard task. At the same time, systems tend to be continuously monitored via advanced sensor systems. This data describes the components' failure behaviour and can be exploited for failure diagnosis and learning of reliability models. This paper presents an effective algorithm for the learning of fault trees from data. Fault trees (FTs) are a widespread formalism in reliability engineering. They capture the failure behaviour of components and their propagation through an entire system. To that end, we first use machine learning to compute a Bayesian Network (BN) highlighting probabilistic relationships between the failures of components and root causes. Then, we apply a set of rules to translate a BN into an FT, based on the Conditional Probability Tables to decide, amongst others, the nature of gates in the FT. We evaluate our method on synthetic data and a benchmark set of FTs.

4.1 Introduction

Reliability engineering provides methods, tools and techniques to evaluate and mitigate the risks related to complex systems. Fault tree analysis is one of the most prominent technique in this field and is widely deployed in the automotive, aerospace and nuclear industry.

Fault trees (FTs) [143] are deterministic graphical models that represent how component failures arise and propagate through the system, leading to system-level failures. Component failures are modelled in the leaves of the tree as *basic events*. FT *gates* model how combinations of basic events lead to a system failure, represented by the top event in the FT. The analysis of such FTs [127] is multifold: they can be used to compute dependability metrics such as system reliability and availability; understand how systems can fail; identify the best ways to reduce the risk of system failure, etc.

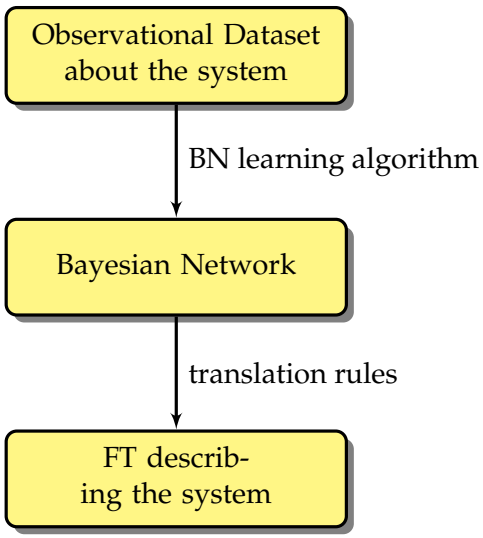


FIGURE 4.1: FT learning framework.

A key bottleneck in FT analysis is, however, the effort needed to construct a faithful FT model. FTs are usually built manually by domain experts. Given the complexity of today's systems, industrial FTs often contain thousands of gates. Hence, their construction is an arduous task, and also error-prone, since their soundness and completeness largely depends on domain expertise.

With the emergence of the industrial Internet-of-Things, Cyber-physical systems are more and more equipped with smart sensor systems, monitoring whether or not a system component is in a failed state or not. Their data can, therefore, be very fruitfully exploited to learn reliability models. Such data can be crucial for the engineers to build an FT [73]. Recent work focused on learning FTs from observational data, identifying causalities from data [105]. In this paper, we focus on FT generation from data, using the prior identification of Bayesian Networks (BNs).

Bayesian Networks [115] are probabilistic graphical models widely used in industry and healthcare as a diagnosis tool. BNs are also used in dependability analysis of systems, being an interesting formalism for the generation of qualitative and diagnosis measurements. In this case, given an FT representing a system's failure behaviour, a set of rules are applied to transform this FT into a BN, to generate these metrics later [22].

This methodology caught our attention: considering a case where data is available, but the FT structure is unavailable mostly due to lack of expert knowledge, the reverse operation seems needed. We considered the case where observational data is available, and where the basic and intermediate events, i.e. observed variables, are also known. Learning the structure as well as the parameters of a BN can be optimally performed from data [43]. Given the learned BN, we apply a set of translation rules to translate any BN node (a random variable) into an FT gate (a deterministic OR or AND gate), using the probabilistic information from the local conditional probability tables inferred from data. Our framework is depicted in Fig. 4.1.

We applied our algorithm, both on synthetic data and a benchmark of FTs previously studied in the literature. Our experiments show that our method is efficient and accurate ($> 99\%$).

Being a first step, our algorithm focuses on *static* FTs, featuring only Boolean gates. An important topic for future work is the extension to dynamic FTs [51]. Therefore, we will discuss an extension to the learning of dynamic BNs, that can be, in turn, translated into dynamic FTs.

This chapter is organized as follows. Section 4.2 reviews related work on learning FTs from data. In Section 4.3 we recall preliminary definitions on FTs and BNs. We present then in Section 4.4 our technique to infer an FT using BN translation. In Section 4.5 we show the results we achieved. Finally, we discuss and conclude about future work.

4.2 Related Work

Related work on learning FTs spans several areas of research: recent work on the generation of FTs from observational data describing the system; since FTs are in essence Boolean functions, the literature on learning Boolean functions from observational data; and, finally, relations between BNs and FTs.

Learning Boolean formulas and classifiers from data. Observational data was used to generate FTs with the IFT algorithm [96] based on standard decision-tree statistical learning. The advantage of learning a graphical decision tree out of data is the inherent interpretability of decision-tree models and their ease of translation into other graphical models. Boolean formulas or networks are learnt using a similar tree-based method [77]. The classic C4.5 learning algorithm [118] yields a Boolean decision tree that is easily translatable into a Boolean formula, by constructing the conjunction of all paths leading to a leaf modelling a True value (i.e., system failure), and then simplifying the Boolean function (by either computing its conjunctive or disjunctive normal form). The resulting models encode the same information as a decision tree (i.e., a classifier for the observational data), so they lack the validation of causal relations, but are expected to preserve their predictive power about the system. Furthermore, Boolean formulas were also machine-learnt using black-box classifiers (classifiers not easily interpretable as a graphical model). Such methods include Support-Vector Machines [129], Logistic Regression [16] and Naive Bayes [128].

Learning causal models from data. LIFT [105] is a recent approach for learning static FTs from data, with Boolean event variables, n-ary *And*/*Or* gates, annotated with event failure probabilities. All intermediate events to be included in the FT must be present in the dataset, but not all may be needed in the FT. LIFT also includes a causal validation step to filter for the most likely causal relationships among system events, but the worst-case complexity is exponential in the number of system events in the data. Its main advantage is that of being one of the few

automated FT-learning methods which specifically validate causality. However, LIFT is sensitive to noise, which is a drawback when dealing with real-world data.

Bayesian Networks [87] are standard graphical models which can be learnt from data. They are widely used in industry, as well as in the health domain as a diagnosis tool. These models have straightforward translations into FTs [21, 22], which are used in FT analysis in order to compute metrics such as reliability or expected number of failures. It has been shown that BNs are hard to synthesize accurately [40]. However, assumptions provided by domain experts can reduce the search space for uncovering these models from data. While previous study [79] favour BNs due to their more general formalism, both in terms of uncertainty as in nominality of data variables, we stand up for FTs, due to their intuitive formalism preferred in industry. We propose a novel method to learn an FT from a tabular dataset composed of observational tuples, in which values (failures) for each Boolean basic event, intermediate event and top event in the system are known. We compare it with a subset of these existing learning algorithm, in terms of performance when fitting data.

4.3 Background

In this section, we first define the structure of FTs, consisting of logic gates and Boolean fault events (an event has value True if that fault occurs in the system). *Intermediate events* (IEs) in the FT are logical combinations of other events (i.e. the OR or AND of a set of events), with only *basic events* (BEs) as the leaves of the tree, and one special intermediate event called the *top event* as root. We also formally define an observational dataset from which we learn an FT. The *top event* of the FT must be a variable in the dataset and can be seen as the outcome to predict by the FT. We finally describe Bayesian Networks, which are graphical models representing probabilistic dependencies between variables. The formulations below follow definitions from [131] and [105].

4.3.1 Fault Trees

Fault trees model how component failures propagate into system failures. Subtrees can be shared so that FTs can be directed acyclic graphs (DAGs). The leaves of the tree model component failures and are called *basic events*. Two types of gates (*And* and *Or*) model how BE failures lead to system failures. The graphical notation for gates and basic events was previously given in Fig. 3.1.

Definition 4.3.1. *A **gate** G is a tuple $(t, \boldsymbol{I}, O)$ s.t.:*

- *t is the type of G with $t \in \{And, Or\}$.*
- *$\boldsymbol{I}$ is a set of $n \geq 2$ input events of G.*
- *O is the event that is the output of G.*

We denote by $I(G)$ the set of events in the input of G and by $O(G)$ the event in the output of G.

Definition 4.3.2. *An And gate is a gate $(And, \boldsymbol{I}, O)$ where output O occurs if and only if every $i \in \boldsymbol{I}$ occurs. An Or gate is a gate $(Or, \boldsymbol{I}, O)$ where output O occurs if and only if at least one $i \in \boldsymbol{I}$ occurs.*

Definition 4.3.3. *A **basic event** B is an event with no input and with a failure rate denoted p_B.*

Other gates are also used: the XOR (exclusive OR), voting, and NOT gates [143, 83].

The root of the tree is the top event T, which represents the failure condition of interest, such as the stranding of a train, or the unplanned unavailability of a satellite. An FT fails if its top event fails.

Definition 4.3.4. *A **fault tree $\boldsymbol{F}$** is a tuple $(\boldsymbol{BE}, \boldsymbol{IE}, T, \boldsymbol{G})$ where:*

- *$\boldsymbol{BE}$ is the set of basic events.*
- *$\boldsymbol{IE}$ is the set of intermediate events.*
- *T is the top event, $T \in \boldsymbol{IE}$.*
- *$\boldsymbol{G}$ is the set of gates; $\forall G \in \boldsymbol{G}, I(G) \subset \boldsymbol{IE} \cup \boldsymbol{BE}, O(G) \in \boldsymbol{IE}$.*
- *The graph formed by $\boldsymbol{G}$ should be connected and acyclic, with the top event T as unique root.*

4.3.2 Bayesian Networks

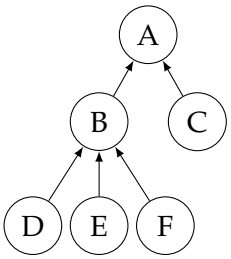
Bayesian networks (BNs) are graphical models where nodes represent random variables and arrows represent probabilistic dependencies between them. In the remaining of this chapter, we consider the case of binary *discrete* BNs where variables are Boolean.

The graphical structure $G = (\mathbf{V}, A)$ of a Bayesian network is a *directed acyclic graph* (DAG), where $\mathbf{V} = \mathbf{BE} \cup \mathbf{IE}$ is the set of *nodes* and A is the set of *edges*. We say that if a node has a *parent* node, then the node is marginally dependent (or unconditionally dependent) of the parent node. A node is said to be a *root* node when it has no parent node. Each node in the BN structure is associated

with a Conditional Probability Table (CPT). The joint distribution of the BN can be factorized according to its graphical structure, which leads to:

$$P(V_1, \ldots, V_n) = \prod_{i=1}^{n} P(V_i \mid \Pi_{V_i}) \tag{4.1}$$

where Π_{V_i} denotes the set of parents of the node V_i in the graph G. In such a case, both the joint and the local distributions (i.e. the CPTs) are multinomial.



(A) Example BN.

		$P(A \mid B, C)$	
		$A = 0$	$A = 1$
$B = 0$	$C = 0$	1	0
$B = 0$	$C = 1$	0	1
$B = 1$	$C = 0$	0	1
$B = 1$	$C = 1$	0	1

(B) Example CPT for variable A.

			$P(B \mid D, E, F)$	
			$B = 0$	$B = 1$
$D = 0$	$E = 0$	$F = 0$	1	0
$D = 0$	$E = 0$	$F = 1$	1	0
$D = 0$	$E = 1$	$F = 0$	1	0
$D = 0$	$E = 1$	$F = 1$	1	0
$D = 1$	$E = 0$	$F = 0$	1	0
$D = 1$	$E = 0$	$F = 1$	1	0
$D = 1$	$E = 1$	$F = 0$	1	0
$D = 1$	$E = 1$	$F = 1$	0	1

(C) Example CPT for variable B.

FIGURE 4.2: Example BN and related CPTs.

An example BN and related Conditional Probability Tables (CPTs) is shown in Fig. 4.2. Note here that these CPTs are a case of *certainty*, where the conditional probabilities are set to either 0 or 1 (unlike CPTs with uncertainty, where probabilities range from 0 to 1).

4.3.3 Learning Dataset

We define a dataset as a collection of records. Each record is a valuation for the set of BEs and IEs, indicating whether a *failure* was observed for that BE/IE. Further, we assume that our dataset is *labeled*, i.e., also indicates whether the top event T has failed, yielding the outcome of the FT.

Definition 4.3.5. *A record R over the set of variables $\boldsymbol{BE} \cup \boldsymbol{IE}$ is a list containing tuples $[(V_i, v_i)], \forall V_i \in \boldsymbol{BE} \cup \boldsymbol{IE}$, where v_i is a Boolean value of V_i.*

We call a *noisy* record one where the value of at least one variable has been measured incorrectly.

Definition 4.3.6. *A dataset $\boldsymbol{D}$ is a set of r records, all over the same set of variables $\boldsymbol{BE} \cup \boldsymbol{IE}$. Each variable name in $\boldsymbol{BE} \cup \boldsymbol{IE}$ forms a column in $\boldsymbol{D}$, and each record forms a row.*

Table 4.1 shows an example dataset for the BN from Fig. 4.2a.

A	B	C	D	E	F
0	0	0	0	0	0
0	0	0	1	0	1
1	0	1	0	0	1
1	0	1	1	1	0
0	0	0	1	1	0
1	0	1	0	1	1
0	0	0	0	1	0
1	0	1	0	1	1
0	0	0	0	1	0
1	1	0	1	1	1

TABLE 4.1: Example dataset for BN in Fig. 4.2a.

4.4 Learning Fault Trees from Diagnosis Models

Learning of BNs is two-fold: the first task is to perform *structure learning*, where the goal is to learn the *skeleton* of the BN together with the directions of the dependencies. The second task is *parameter learning*, to approximate CPTs from data (e.g. via maximum likelihood estimation). Structure learning is typically done by heuristic search (using, e.g. the BIC scoring function and a TABU search). It provides approximate Bayesian network structures in a reasonable time, as finding optimal structure is intractable. It is important to note here that involving domain knowledge can reduce the complexity of structure learning: if one knows what dependencies have to be present (or not) in the structure, this given information reduces the search space of the structure.

In this section, we show how we perform efficient structure and parameter learning of BNs from data. We also show how we achieve the translation of a BN into an FT thanks to a set of translations rules.

4.4.1 Learning of Bayesian Networks

Structure Learning BN structure learning algorithms can be grouped in two categories: The first one, *constraint-based algorithms* [23], which learn structure causal models. They can be summarized in two steps: (1) learn the *skeleton* (i.e. the undirected graph underlying the network structure) of the network; (2) set the direction of all the edges. The second category, *score-based algorithms*, assign a score to each candidate BN and try to maximize it with some heuristic search algorithm. Greedy search algorithms (such as *hill-climbing* or *tabu search*) are common.

Blacklisting Prior assumptions on the data or the problem to solve can be integrated into the learning algorithm through *blacklists* and *whitelists*. They define arcs which are respectively missing or present in the BN. In our setting, we defined three families of dependencies which are not desired in the BN structure. These undesired arcs represent conditional dependencies which are assumed not to be relevant in the BN. Our blacklisted arcs are the followings:

- $T \rightarrow V_i, \forall V_i \in \mathbf{BE} \cup \mathbf{IE}$. However, it is allowed that $V_i \rightarrow T$ because this captures the natural cause-effect direction of the system being modeled.
- $B_i \rightarrow B_j, \forall B_i, B_j \in \mathbf{BE}$. Two any BEs are not involved in direct probabilistic interaction among themselves (however, this constraint can be relaxed, see Section 4.6).
- $IE \rightarrow B, \forall IE \in \mathbf{IE}, \forall B \in \mathbf{BE}$. However, it is allowed that $IE \rightarrow B$ because this captures the natural cause-effect direction of the system being modeled. This constraint is particularly necessary for a good identification of which variables will play the role of gates once the BN is translated into an FT.

The blacklisted arcs of BN in Fig. 4.2a are then as follows:

- $A \rightarrow B$
- $A \rightarrow C$
- $A \rightarrow D$
- $A \rightarrow E$
- $A \rightarrow F$
- $C \rightarrow D$
- $C \rightarrow E$
- $C \rightarrow F$
- $D \rightarrow C$
- $D \rightarrow E$
- $D \rightarrow F$
- $E \rightarrow C$
- $E \rightarrow D$
- $E \rightarrow F$
- $F \rightarrow C$
- $F \rightarrow D$
- $F \rightarrow E$
- $B \rightarrow C$
- $B \rightarrow D$
- $B \rightarrow E$
- $B \rightarrow F$

Parameter Learning The goal of parameter learning is to approximate, given data on random variables, the CPTs of a given BN structure. After learning the parameters via maximum likelihood estimation, the BIC (Bayesian Information Criterion) score of the BN is computed. If M is a candidate BN, then its BIC is given by:

$$\mathrm{BIC}(M) = -2\ln(L) + K\ln(N) \tag{4.2}$$

where L is the maximized log-likelihood of M, K is the number of parameters of M, and N is the sample size (i.e. the size of the dataset). Models with smaller BIC are preferred. It measures not only the fitness of the model to the data but also has the advantage of controlling for the model complexity.

4.4.2 Translation Rules

In [21, 22], a two-fold method to translate an FT into a BN was proposed. It consists of a one-to-one transformation of FT gates into event nodes in the BN, as well as the related CPT generation:

1. Translate T and all $V_i \in \mathbf{BE} \cup \mathbf{IE}$ into BN nodes.
2. Add probabilistic dependencies between related events.
3. Set failure probabilities of BEs to follow definitions of p_B for all $B \in \mathbf{BE}$.
4. Set failure probabilities of IEs to follow the truth tables of related events.

In practice, the latter can be replaced based on expert knowledge. This is in order to relax conditions with certainty to CPTs with uncertainty.

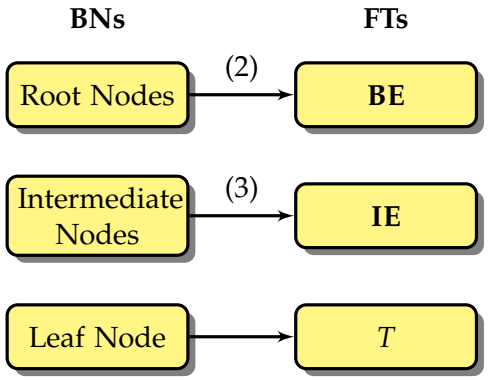


FIGURE 4.3: Translation of a BN into an FT.

In our case, a BN and the associated CPTs are already learned from data. As a consequence, we have to perform the reverse translation compared to [21, 22]. Our translation is described below, and also shown in Fig. 4.3:

1. Remove all nodes which are not connected to the system level failure node in the BN.
2. For each *root* node of the BN, create a BE in the FT (each root node in the BN is necessarily a basic event, following blacklisting constraints). For each created BE B, set p_B to $P(B = 1)$.
3. For each non-root node of the BN, create an IE in the FT.
4. Connect events in the FT as related nodes in the BN are connected.
5. For each gate in the FT, identify its nature (*Or* or *And*) by inspecting the CPTs.

The identification of the nature of a gate follows the CPT of its related node. For each node $V_i \in$ **IE**, we apply the following set of translation rules:

1. if $P(V_i = 1 \mid \bigwedge_{c \in C} c = 1) = 1$ where C is the set of parents of V_i, then V_i is an *And* gate in the FT.
2. $P(V_i = 1 \mid \bigvee_{c \in C} c = 1) = 1$ where C is the set of parents of V_i, then V_i is an *Or* gate in the FT.

Uncertainty One can see here that probabilistic non-root nodes of a BN are translated into deterministic gates in the FT. The certainty feature of the CPTs may be unrealistic in the real world. One may prefer to assume that the logical gates are *noisy* (i.e. probabilistic) and return an FT gate G with an associated probability p_G, as being the probability of failure of the IE when the logical condition is met. Therefore, we defined an alternative uncertain case for translating an FT:

1. if $P(V_i = 1 \mid \bigwedge_{c \in C} c = 1) > 1 - \epsilon$ where C is the set of parents of V_i, then V_i is an *And* gate in the FT with $p_G = P(V_i = 1 \mid \bigwedge_{c \in C} c = 1)$
2. $P(V_i = 1 \mid \bigvee_{c \in C} c = 1) > 1 - \epsilon$ where C is the set of parents of V_i, then V_i is an *Or* gate with $p_G = P(V_i = 1 \mid \bigvee_{c \in C} c = 1)$

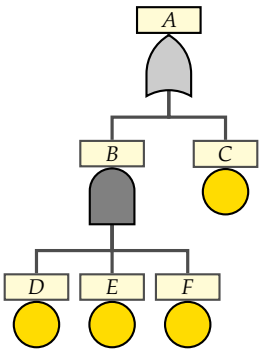
where ϵ is a tolerance factor for deciding whether a gate is an *Or* gate or an *And* gate.

Note here that, since subtrees can be shared in an FT, it may be the case that a BE or an IE is an input to more than one gate.

An example of translation of BN and CPTs in Fig. 4.2 into an FT is shown in Fig. 4.4.

$$P(A = 1 \mid B = 1 \vee C = 1) = 1$$
$$P(B = 1 \mid D = 1 \wedge E = 1 \wedge F = 1) = 1$$

(A) Satisfied/Derived conditions in CPTs.



(B) Induced Fault Tree.

FIGURE 4.4: Translation of BN of Fig. 4.2 into an FT.

4.5 Experimental Evaluation

We have evaluated the efficiency and effectiveness of our method for a set of synthetic cases, as well as for industrial benchmarks. We compared our methods with other learning techniques, among others, five approaches from the literature. The accuracy of an FT is the number of records in the dataset for which the value of the top event, given the values of the BEs, is correctly computed.

4.5.1 Experimental set up

The first three methods in our evaluation are: (1) Support Vector Machine (abbreviated svm in the figures) [129], (2) Logistic Regression (log) [16] and (3) Naive Bayes Classifier (nba) [128]. These methods are Boolean classifiers that, given the values of the BEs, predict the value of the top event T. During our experiments, we used state-of-the-art Python implementations for these techniques [116]. Being classifiers, methods (1) - (3) do not yield fault tree models, only a prediction for the value of T.

Then, we have used three methods that do learn FT models: (4) We have compared our results to the well-known C4.5 algorithm for learning decision trees (abbreviated c45) [118]. Decision trees can be transformed to FTs, by first computing in the decision tree the conjunction of all paths leading to failure leaves, and then simplifying the conjunction to CNF. (5) We have also compared to the earlier LIFT [105] approach, which returns an FT.

To compare these methods, the observational data was divided into 2 sets: one training set, used as input to the learning algorithms (with an average of 2/3 of all possible observations and possibly duplicate records), and a test set containing all observational variables (complete truth table), used to evaluate the solution returned by our algorithms. We used the library *bnlearn* to learn BNs [131]. The parameters of our algorithm were a *tabu search* for learning the structure of the BN, and the maximization of the BIC score for learning the parameters. The blacklisting of edges was set as detailed in Section 4.4.1. We considered the uncertain case, with $\epsilon = 0.1$.

A further experiment was carried out (bn-o) where edges $V_i \rightarrow T$ for all parent node of T in the BN are known and set as *whitelist*, i.e. the top event, and its direct input events in the FT are known. This is a real case in practice: indeed, experts may partially know the structure of the FT (hence of the BN) and like to include this information before the learning of the BN.

4.5.2 Synthetic Dataset

We first considered 100 randomly generated FTs with 6 to 15 BEs and for each FT, a randomly generated data set, with 200 to 230k records. The FTs considered contain on average 5 gates, and BEs probabilities range from 1e-4 to 1e-2. Figures 4.5 and 4.6 present respectively the average accuracy and the average runtime, both as functions of the number of BEs.

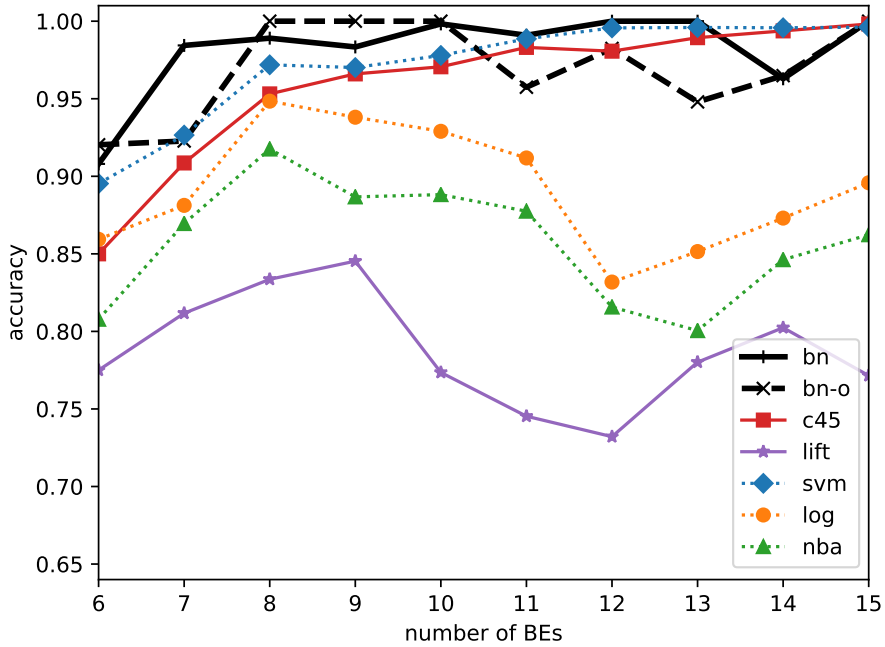


FIGURE 4.5: Accuracy as a function of BEs.

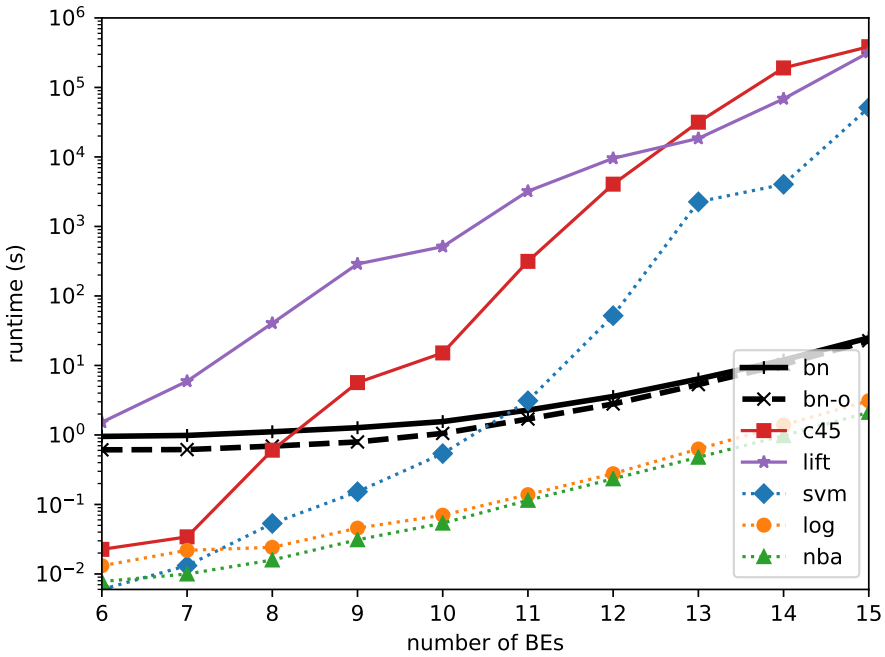


FIGURE 4.6: Runtime of different algorithms.

Fig. 4.5 shows that our methods bn and bn-o are optimal as well in terms of accuracy as runtime. When the number of BEs in the FT is high, svm and c45 are equally optimal. However, the svm method only provides a classifier, not an FT. Further, the c45 method does not perform well in terms of runtime, see Fig. 4.6. Methods log and nba are fast, but provide low accuracy. Finally, we see that LIFT obtains less good results. This can be explained by an exponential complexity, as well as the fact that LIFT return FTs with a low depth in most of the cases or even missing many BEs.

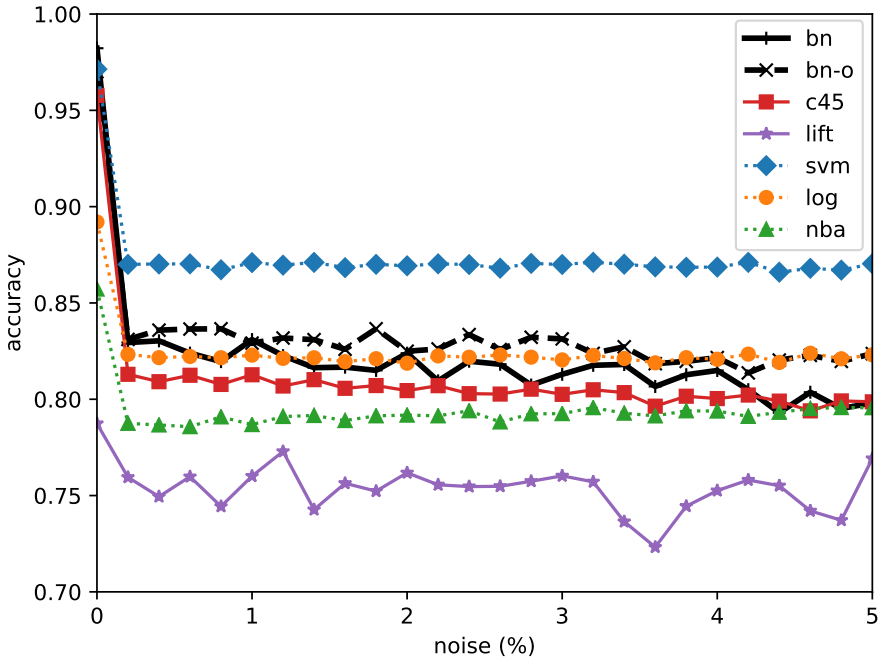


FIGURE 4.7: Effect of noise on the learned FTs.

We also carried out an experiment where noise is added in the dataset, in order to test the robustness of the different algorithms. The noise varies from 0 to 5% of noisy records. In the results shown in Fig. 4.7, we see that our methods are relatively robust against noise compared to other methods. However, we see that the accuracy of the learned FTs drops whenever noise is present in the dataset.

4.5.3 Fault Tree Benchmark

We present here the results we obtained for a set of publicly available benchmark suite[1], consisting of industrial FTs from the literature [25, 74, 100, 102, 135]. The FTs used are Cardiac Assist System (CAS), Container Seal Design Example (CSD), Multiprocessor Computing System (MCS), Monopropellant Propulsion System (MPS), Pressure Tank (PT), Sensor Filter Network (SF14) and Spread Mooring System (SMS_A1). They are thoroughly described in Section 3.6.5. Whereas the

[1]https://dftbenchmarks.utwente.nl/

FT models were given in the literature, no data sets were available. Therefore, we have randomly generated these data sets, containing 10M records per case (in order to handle low failure rates). Since the benchmark does provide failure probabilities per BEs, we have used those probabilities: If p_B is the failure probability of BE B in the benchmark, then we set, in each data record, $R[B] = 1$ with probability p_B. Note that our implementations and dataset are available[2].

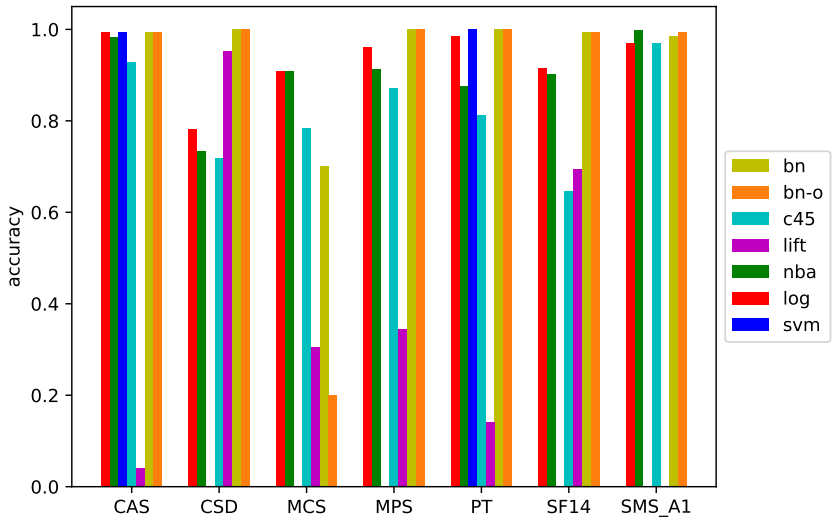


FIGURE 4.8: Results of FT Benchmark (accuracy).

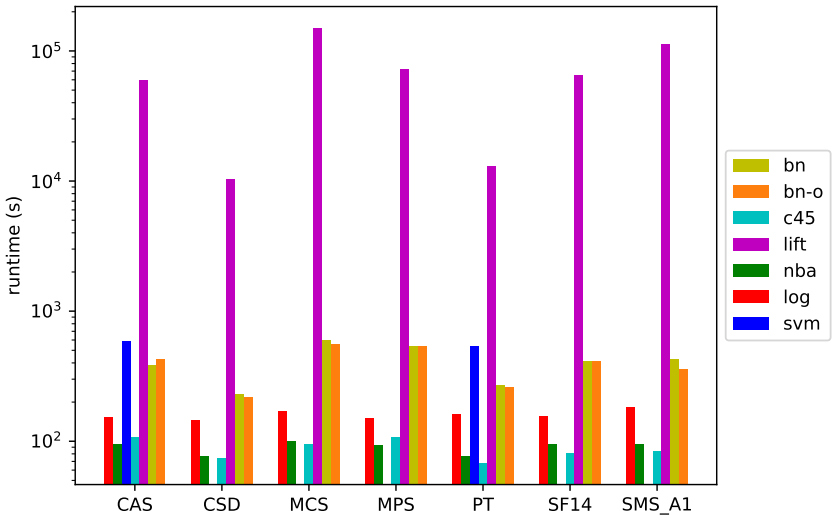


FIGURE 4.9: Results of FT Benchmark (runtime).

[2]https://gitlab.science.ru.nl/alinard/learning-ft-bn

4.6 Discussion

One of the limitations of our work is the need for data exhaustiveness: indeed, we require knowledge about all BEs, as well as all intermediate events. One can imagine a realistic case where some of the intermediate events are missing. However, uncovering these hidden variables is known to be hard in BN synthesis.

Another limitation is the focus on static FTs. However, one can see straightforward extensions that could use the formalism of dynamic FTs (DFTs, [51]). These come with additional gates, catering for common dependability patterns like spare management and functional dependencies. Some additional translation rules lead to semantics missing in static FTs, but present in DFTs:

1. *Functional Dependency* (FDEP): models how the occurrence of an event triggers the failure of other components. So if $P(V_i = 1 \mid \underset{c \in C}{\Longrightarrow} c = 1) = 1$ where C is the set of parents of V_i, then $c \rightarrow V_i, \forall c \in C$ is a FDEP in the DFT.

Also, some constraints relaxation, e.g. in the blacklisting of edges, can lead to DFT gates:

2. *Rate Dependency* (RDEP): models how a BE failure influences other BE failures. When the blacklisting of edges $B_i \rightarrow B_j, \forall B_i, B_j \in \mathbf{BE}$ is relaxed, then edge $B_i \rightarrow B_j$ is a RDEP.

Another interesting formalism to take into account is the one of dynamic BNs (DBNs): they relate variables to each other over time. It is known that DBNs can model, among others, priority AND gates of DFTs [127]. We leave the translation of DBNs into DFTs for future work.

4.7 Conclusion

We presented an attractive method for the automated generation of FTs from precomputed BNs from data. To that extent, we defined a set of translation rules from one formalism to another. Our results show that our method is particularly robust to noise. We have also applied our technique to a benchmark of FTs to ensure its practical relevance. Our future research will focus on the learning of dynamic FTs. This will be done by the prior computation of a dynamic BN from time series, and the translation of such a dynamic BN to a dynamic FT with appropriate rules.

Chapter 5

Learning Unions of k-Testable Languages

A classical problem in grammatical inference is to identify a language from a set of examples. In this chapter, we address the problem of identifying a union of languages from examples that belong to several *different* unknown languages. Indeed, decomposing a language into smaller pieces that are easier to represent should make learning easier than aiming for a too generalized language. In particular, we consider k-testable languages in the strict sense (k-TSS). These are defined by a set of allowed prefixes, infixes (sub-strings) and suffixes that words in the language may contain. We establish a Galois connection between the lattice of all languages over an alphabet Σ, and the lattice of k-TSS languages over Σ. We also define a simple metric on k-TSS languages. The Galois connection and the metric allow us to derive an efficient algorithm to learn the union of k-TSS languages. We evaluate our algorithm on an industrial dataset and thus demonstrate the relevance of our approach.

5.1 Introduction

A common problem in grammatical inference is to find, i.e. *learn*, a regular language from a set of examples of that language. When this set is divided into positive examples (belonging to the language) and negative examples (not belonging to the language), the problem is typically solved by searching for the smallest deterministic finite automaton (DFA) that accepts the positive examples and rejects the negative ones. Moreover there exist algorithms which *identify in the limit* a DFA, that is, they eventually learn correctly any language/automaton from such examples [63].

We consider in this work a setting where one can observe positive examples from multiple different languages, but they are given together, and it is not clear to which language each example belongs. For example, given the following set of strings $S = \{aa, aaa, aaaa, abab, ababab, abba, abbba, abbbba\}$, learning a single automaton will be less informative than learning several DFA encoding respectively the languages a^*, $(ab)^*$ and ab^*a. There is a trade-off between the number of languages and how specific each language should be. That is, covering all words through a single language may not be the desired result, but having a language

for each word may also not be desired. The problem at hand is therefore double: to cluster the examples and learn the corresponding languages.

In this chapter, we focus on k-testable languages in the strict sense (k-TSS) [99]. A k-TSS language is determined by a finite set of substrings of length at most k that are allowed to appear in the strings of the language. It has been proved that, unlike for regular languages, algorithms can learn k-TSS languages in the limit from text [149]. Practically, this learning guarantee has been used in a wide range of applications [17, 44, 122, 138]. However, all these applications consider learning of a sole k-TSS language [17], or the training of several k-TSS languages in a context of supervised learning [138]. Learning unions of k-TSS languages has been suggested in [139].

The first contribution of this chapter is a Galois connection between the lattice of all languages over alphabet Σ and the lattice of k-TSS languages over Σ. This result provides a unifying and abstract perspective on known properties of k-TSS languages, but also leads to several new insights. The Galois connection allows to give an alternative proof of the learnability in the limit of k-TSS languages and suggests an algorithm for learning unions of k-TSS languages. A second contribution is the definition of a simple metric on k-TSS languages. Based on this metric, we define a clustering algorithm that allows us to efficiently learn unions of k-TSS languages.

Our research was initially motivated by a case study of print jobs that are submitted to large industrial printers. Strings of symbols can represent these print jobs, where each symbol denotes a different media type, such as a book cover or a newspaper page. Together, this set of print jobs makes for a fairly complicated 'language'. Nevertheless, we observed that each print job could be classified as belonging to one of a fixed set of categories, such as 'book' or 'newspaper'. Two print jobs that belong to the same category are typically similar, to the extent that they only differ in terms of prefixes, infixes and suffixes. Therefore, the languages stand for the different families of print jobs. Our goal is to uncover these k-TSS languages.

This chapter is organized as follows. In Section 5.2 we recall preliminary definitions on k-TSS languages and define a Galois connection that characterizes these languages. We then present in Section 5.3 our algorithm for learning unions of k-TSS languages. Finally, we report on the results we achieved for the industrial case study in Section 5.4.

5.2 k-Testable Languages

The class of k-testable languages in the strict sense (k-TSS) has been introduced by [99]. Informally, a k-TSS language is determined by a finite set of substrings of length at most k that are allowed to appear in the strings of the language. This makes it possible to use as a parser a sliding window of size k, which rejects the strings that at some point, do not comply with the conditions. Concepts related to k-TSS languages have been widely used, e.g. in information theory, pattern recognition and DNA sequence analysis [58, 149]. Several definitions of k-TSS

languages occur in the literature, but the differences are technical. In this section, we present a slight variation of the definition of k-TSS languages from [70], which in turn is a variation of the definition occurring in [58, 59]. We establish a Galois connection that characterizes k-TSS languages and show how this Galois connection may be used to infer a learning algorithm.

We write $\mathbb{N}$ to denote the set of natural numbers, and let i, j, k, m, and n range over $\mathbb{N}$.

5.2.1 Strings

Throughout this chapter, we fix a finite set Σ of *symbols*. A *string* $x = a_1 \dots a_n$ is a finite sequence of symbols. The *length* of a string x, denoted $\mid x \mid$ is the number of symbols occurring in it. The empty string is denoted λ. We denote by Σ^* the set of all strings over Σ, and by Σ^+ the set of all nonempty strings over Σ (i.e. $\Sigma^* = \Sigma^+ \cup \{\lambda\}$). Similarly, we denote by $\Sigma^{<i}$, Σ^i and $\Sigma^{>i}$ the sets of strings over Σ of length less than i, equal to i, and greater than i, respectively.

Given two strings u and v, we will denote by $u \cdot v$ the concatenation of u and v. When the context allows it, $u \cdot v$ shall be simply written uv. We say that u is a *prefix* of v iff there exists a string w such that $uw = v$. Similarly, u is a *suffix* of v iff there exists a string w such that $wu = v$. We denote by $x[:k]$ the prefix of length k of x and $x[-k:]$ the suffix of length k of x.

A *language* is any set of strings, so, therefore, a subset of Σ^*. Concatenation is lifted to languages by defining $L \cdot L' = \{u \cdot v \mid u \in L \text{ and } v \in L'\}$. Again, we will write LL' instead of $L \cdot L'$ when the context allows it.

5.2.2 k-Testable Languages

A k-TSS language is determined by finite sets of strings of length $k-1$ or k that are allowed as prefixes, suffixes and substrings, respectively, together with all the short strings (with length at most $k-1$) contained in the language. The finite sets of allowed strings are listed in what [99] called a *k-test vector*. The following definition is taken from [70], except that we have omitted the fixed alphabet Σ as an element in the tuple, and added a technical condition ($I \cap F = C \cap \Sigma^{k-1}$) that we need to prove Theorem 5.2.7.

Definition 5.2.1. *Let $k > 0$. A* k-test vector *is a 4-tuple $Z = \langle I, F, T, C \rangle$ where*

- $I \subseteq \Sigma^{k-1}$ *is a set of allowed* prefixes,
- $F \subseteq \Sigma^{k-1}$ *is a set of allowed* suffixes,
- $T \subseteq \Sigma^k$ *is a set of allowed* segments, *and*
- $C \subseteq \Sigma^{<k}$ *is a set of allowed* short strings *satisfying* $I \cap F = C \cap \Sigma^{k-1}$.

We write $\mathcal{T}_k$ for the set of k-test vectors.

Note that the set $\mathcal{T}_k$ of k-test vectors is finite. We equip set $\mathcal{T}_k$ with a partial order structure as follows.

Definition 5.2.2. *Let $k > 0$. The relation $\sqsubseteq$ on $\mathcal{T}_k$ is given by*

$$\langle I, F, T, C\rangle \sqsubseteq \langle I', F', T', C'\rangle \;\Leftrightarrow\; I \subseteq I' \text{ and } F \subseteq F' \text{ and } T \subseteq T' \text{ and } C \subseteq C'.$$

With respect to this ordering, $\mathcal{T}_k$ has a least element $\bot = \langle \emptyset, \emptyset, \emptyset, \emptyset\rangle$ and a greatest element $\top = \langle \Sigma^{k-1}, \Sigma^{k-1}, \Sigma^k, \Sigma^{<k}\rangle$. The union, intersection *and* symmetric difference *of two k-test vectors $Z = \langle I, F, T, C\rangle$ and $Z' = \langle I', F', T', C'\rangle$ are given by, respectively,*

$$\begin{aligned} Z \sqcup Z' &= \langle I \cup I', F \cup F', T \cup T', C \cup C' \cup (I \cap F') \cup (I' \cap F)\rangle, \\ Z \sqcap Z' &= \langle I \cap I', F \cap F', T \cap T', C \cap C'\rangle, \\ Z \mathbin{\triangle} Z' &= \langle I \mathbin{\triangle} I', F \mathbin{\triangle} F', T \mathbin{\triangle} T', C \mathbin{\triangle} C' \mathbin{\triangle} (I' \cap F) \mathbin{\triangle} (I \cap F')\rangle. \end{aligned}$$

The reader may check that $Z \sqcup Z'$, $Z \sqcap Z'$ and $Z \mathbin{\triangle} Z'$ are k-test vectors indeed, preserving the property $I \cap F = C \cap \Sigma^{k-1}$. The reader may also check that $(\mathcal{T}_k, \sqsubseteq)$ is a lattice with $Z \sqcup Z'$ the least upper bound of Z and Z', and $Z \sqcap Z'$ the greatest lower bound of Z and Z'. The symmetric difference operation $\mathbin{\triangle}$ will be used further on to define a metric on k-test vectors.

We can associate a k-test vector $\alpha_k(L)$ to each language L by taking all prefixes of length $k-1$ of the strings in L, all suffixes of length $k-1$ of the strings in L , and all substrings of length k of the strings in L. Any string which is both an allowed prefix and an allowed suffix is also a short string, as well as any string in L with length less than $k-1$.

Definition 5.2.3. *Let $L \subseteq \Sigma^*$ be a language and $k \in \mathbb{N}$. Then $\alpha_k(L)$ is the k-test vector $\langle I_k(L), F_k(L), T_k(L), C_k(L)\rangle$ where*

- $I_k(L) = \{u \in \Sigma^{k-1} \mid \exists v \in \Sigma^* : uv \in L\}$,
- $F_k(L) = \{w \in \Sigma^{k-1} \mid \exists v \in \Sigma^* : vw \in L\}$,
- $T_k(L) = \{v \in \Sigma^k \mid \exists u, w \in \Sigma^* : uvw \in L\}$, *and*
- $C_k(L) = (L \cap \Sigma^{<k-1}) \cup (I_k(L) \cap F_k(L))$.

It is easy to see that operation $\alpha_k : 2^{\Sigma^*} \to \mathcal{T}_k$ is monotone.

Proposition 5.2.4. *For all languages L, L' and for all $k > 0$,*

$$L \subseteq L' \;\Rightarrow\; \alpha_k(L) \sqsubseteq \alpha_k(L').$$

Conversely, we associate a language $\gamma_k(Z)$ to each k-test vector $Z = \langle I, F, T, C\rangle$, consisting of all the short strings from C together with all strings of length at least $k-1$ whose prefix of length $k-1$ is in I, whose suffix of length $k-1$ is in F, and where all substrings of length k belong to T.

Definition 5.2.5. *Let $Z = \langle I, F, T, C\rangle$ be a k-test vector, for some $k > 0$. Then*

$$\gamma_k(Z) \;=\; C \cup ((I\Sigma^* \cap \Sigma^* F) \setminus (\Sigma^*(\Sigma^k \setminus T)\Sigma^*)).$$

*We say that a language L is k-*testable in the strict sense (k-TSS) *if there exists a k-test vector Z such that $L = \gamma_k(Z)$. Note that all k-TSS languages are regular.*

Again, it is easy to see that operation $\gamma_k : \mathcal{T}_k \to 2^{\Sigma^*}$ is monotone.

Proposition 5.2.6. *For all $k > 0$ and for all k-test vectors Z and Z',*

$$Z \sqsubseteq Z' \;\Rightarrow\; \gamma_k(Z) \subseteq \gamma_k(Z').$$

The next theorem, which is our main result about k-testable languages, asserts that α_k and γ_k form a (monotone) Galois connection [106] between lattices $(\mathcal{T}_k, \sqsubseteq)$ and $(2^{\Sigma^*}, \subseteq)$.

Theorem 5.2.7 (Galois connection)**.** *Let $k > 0$, let $L \subseteq \Sigma^*$ be a language, and let Z be a k-test vector. Then $\alpha_k(L) \sqsubseteq Z \;\Leftrightarrow\; L \subseteq \gamma_k(Z)$.*

Proof. Let $Z = \langle I, T, F, C\rangle$.

$\Rightarrow$. Assume $\alpha_k(L) \sqsubseteq Z$ and $w \in L$. We need to show that $w \in \gamma_k(Z)$. Since $\alpha_k(L) \sqsubseteq Z$ we know that $I_k(L) \subseteq I$, $F_k(L) \subseteq F$, $T_k(L) \subseteq T$ and $C_k(L) \subseteq C$. If $|\, w \,| < k-1$ then $w \in C_k(L) \subseteq C \subseteq \gamma_k(Z)$, and we are done. If $|\, w \,| = k-1$ then $w \in I_k(L)$ and $w \in F_k(L)$. This implies that $w \in C_k(L)$. Now we use again that $C_k(L) \subseteq C \subseteq \gamma_k(Z)$, and we are done. If $|\, w \,| \geq k$, then $I_k(L) \subseteq I$ implies that the prefix of length $k-1$ of w is in I, $F_k(L) \subseteq F$ implies that the suffix of length $k-1$ of w is in F, and $T_k(L) \subseteq T$ implies that any substring of length k of w is in T. Thus $w \in (I\Sigma^* \cap \Sigma^* F) \setminus (\Sigma^*(\Sigma^k \setminus T)\Sigma^*)$ and this implies $w \in \gamma_k(Z)$, as required.

$\Leftarrow$. Assume $L \subseteq \gamma_k(Z)$. We need to show that $\alpha_k(L) \sqsubseteq Z$. This is equivalent to showing that $I_k(L) \subseteq I$, $F_k(L) \subseteq F$, $T_k(L) \subseteq T$ and $C_k(L) \subseteq C$.

- $I_k(L) \subseteq I$. Suppose $u \in I_k(L)$. Then $|\, u \,| = k-1$ and there exists a string v such that $uv \in L$. By the assumption, $uv \in \gamma_k(Z)$. We consider two cases: $v = \lambda$ and $v \neq \lambda$. If $v = \lambda$ then $u \in \gamma_k(Z)$. Since $|\, u \,| = k-1$, this means that $u \in C \cup (I \cap F)$. Since Z is a k-test vector, $I \cap F = C \cap \Sigma^{k-1}$. We can now infer $u \in I \cap F$. Thus, in particular, $u \in I$, as required. If $v \neq \lambda$ then, using $uv \in \gamma_k(Z)$, we conclude that $uv \in I\Sigma^*$. This implies $u \in I$, as required.

- $F_k(L) \subseteq F$. Suppose $w \in F_k(L)$. Then $|\, w \,| = k-1$ and there exists a string v such that $vw \in L$. By the assumption, $vw \in \gamma_k(Z)$. We consider two cases: $v = \lambda$ and $v \neq \lambda$. If $v = \lambda$ then $w \in \gamma_k(Z)$. Since $|\, w \,| = k-1$, this means that $w \in C \cup (I \cap F)$. Since $I \cap F = C \cap \Sigma^{k-1}$, we conclude $w \in I \cap F$. Thus, in particular, $w \in F$, as required. If $v \neq \lambda$ then, using $vw \in \gamma_k(Z)$, we conclude that $vw \in \Sigma^* F$. This implies $w \in F$, as required.

- $T_k(L) \subseteq T$. Suppose $v \in T_k(L)$. Then $\mid v \mid = k$ and there exist strings u and w such that $uvw \in L$. By the assumption, $uvw \in \gamma_k(Z)$. As the length of uvw is at least k, $uvw \in (I\Sigma^* \cap \Sigma^* F) \setminus (\Sigma^*(\Sigma^k \setminus T)\Sigma^*)$. Thus $uvw \notin (\Sigma^*(\Sigma^k \setminus T)\Sigma^*)$. Therefore, uvw does not contain a substring of length k that is not in T. But this implies $v \in T$, as required.
- $C_k(L) \subseteq C$. Suppose $w \in C_k(L)$. Then $w \in L \cap \Sigma^{<k-1}$ or $w \in I_k(L) \cap F_k(L)$. If $w \in L \cap \Sigma^{<k-1}$ then by the assumption $w \in \gamma_k(Z)$, and, as the length of w is less than $k-1$, we conclude $w \in C$, as required. If $w \in I_k(L) \cap F_k(L)$ then $w \in I_k(L)$ and $w \in F_k(L)$. By the previous items it then follows that $w \in I$ and $w \in F$, and therefore $w \in I \cap F$. Now we use $I \cap F = C \cap \Sigma^{k-1}$ to conclude $w \in C$.

□

The above theorem generalizes results on strictly k-testable languages from [58, 149]. Composition $\gamma_k \circ \alpha_k$ is commonly called the *associated closure operator*, and composition $\alpha_k \circ \gamma_k$ is known as the *associated kernel operator*. The fact that we have a Galois connection has some well-known consequences for these associated operators.

Corollary 5.2.7.1. *For all $k > 0$, $\gamma_k \circ \alpha_k$ and $\alpha_k \circ \gamma_k$ are monotone and idempotent.*

Monotony of $\gamma_k \circ \alpha_k$ was established previously as Theorem 3.2 in [58] and as Lemma 3.3 in [149].

Corollary 5.2.7.2. *For all $k > 0$, $L \subseteq \Sigma^*$ and $Z \in \mathcal{T}_k$,*

$$\alpha_k \circ \gamma_k(Z) \sqsubseteq Z \tag{5.1}$$
$$L \subseteq \gamma_k \circ \alpha_k(L) \tag{5.2}$$

Inequality (5.1) asserts that the associated kernel operator $\alpha_k \circ \gamma_k$ is *deflationary*, while inequality (5.2) says that the associated closure operator $\gamma_k \circ \alpha_k$ is *inflationary* (or *extensive*). Inequality (5.2) was established previously as Lemma 3.1 in [58] and (also) as Lemma 3.1 in [149].

Another immediate corollary of the Galois connection is that $\gamma_k \circ \alpha_k(L)$ is the smallest k-TSS language that contains L. This has been established previously as Theorem 3.1 in [58].

Corollary 5.2.7.3. *For all $k > 0$, $L \subseteq \Sigma^*$, and $Z \in \mathcal{T}_k$,*

$$L \subseteq \gamma_k(Z) \quad \Rightarrow \quad \gamma_k \circ \alpha_k(L) \subseteq \gamma_k(Z).$$

As a final corollary, we mention that $\alpha_k \circ \gamma_k(Z)$ is the smallest k-test vector that denotes the same language as Z. This is essentially Lemma 1 of [149].

Corollary 5.2.7.4. *For all $k > 0$ and $Z \in \mathcal{T}_k$, $\gamma_k \circ \alpha_k \circ \gamma_k(Z) = \gamma_k(Z)$. Moreover, for any $Z' \in \mathcal{T}_k$,*

$$\gamma_k(Z) = \gamma_k(Z') \quad \Rightarrow \quad \alpha_k \circ \gamma_k(Z) \sqsubseteq Z'.$$

We can provide a simple characterization of $\alpha_k \circ \gamma_k(Z)$ as the k-test vector obtained by removing all the allowed prefixes, suffixes and segments that do not occur in the k-testable language generated by Z.

Definition 5.2.8. *Let $Z = \langle I, F, T, C \rangle$ be a k-test vector, for some $k > 0$. We say that $u \in I$ is a* junk prefix *of Z if u does not occur as a prefix of any string in $\gamma_k(Z)$. Similarly, we say that $u \in F$ is a* junk suffix *of Z if u does not occur as a suffix of any string in $\gamma_k(Z)$, and we say that $u \in T$ is a* junk segment *of Z if u does not occur as a substring of any string in $\gamma_k(Z)$. We call Z* canonical *if it does not contain any junk prefixes, junk suffixes, or junk segments.*

Proposition 5.2.9. *Let Z be a k-test vector, for some $k > 0$, and let Z' be the canonical k-test vector obtained from Z by deleting all junk prefixes, junk suffixes, and junk segments. Then $\alpha_k \circ \gamma_k(Z) = Z'$.*

5.2.3 Learning k-TSS Languages

It is well-known that any k-TSS language can be identified in the limit from positive examples [58, 59]. Below we recall the basic argument; we refer to [58, 59, 149] for efficient algorithms.

Theorem 5.2.10. *Any k-TSS language can be identified in the limit from positive examples.*

Proof. Let L be a k-TSS language and let $w_1, w_2, w_3, \ldots$ be an enumeration of L. Let $L_0 = \emptyset$ and $L_i = L_{i-1} \cup \{w_i\}$, for $i > 0$. We then have

$$L_1 \subseteq L_2 \subseteq L_3 \subseteq \cdots$$

By monotonicity of α_k (Proposition 5.2.4) we obtain

$$\alpha_k(L_1) \sqsubseteq \alpha_k(L_2) \sqsubseteq \alpha_k(L_3) \sqsubseteq \cdots \tag{5.3}$$

and by monotonicity of γ_k (Proposition 5.2.6)

$$\gamma_k \circ \alpha_k(L_1) \subseteq \gamma_k \circ \alpha_k(L_2) \subseteq \gamma_k \circ \alpha_k(L_3) \subseteq \cdots \tag{5.4}$$

Since $\gamma_k \circ \alpha_k$ is inflationary (Corollary 5.2.7.2), L is a k-TSS language and, for each i, $\gamma_k \circ \alpha_k(L_i)$ is the smallest k-TSS language that contains L_i (Corollary 5.2.7.3), we have

$$L_i \subseteq \gamma_k \circ \alpha_k(L_i) \subseteq L \tag{5.5}$$

Because $(\mathcal{T}_k, \sqsubseteq)$ is a finite partial order it does not have an infinite ascending chain. This means that sequence (5.3) converges. But then sequence (5.4) also converges, that is, there exists an n such that, for all $m \geq n$, $\gamma_k \circ \alpha_k(L_m) = \gamma_k \circ \alpha_k(L_n)$. By equations (5.4) and (5.5) we obtain, for all i,

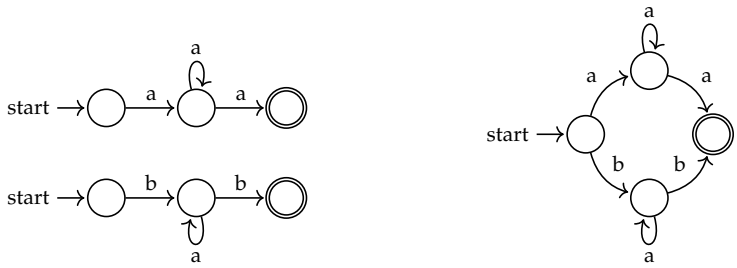
$$L_i \subseteq \gamma_k \circ \alpha_k(L_i) \subseteq \gamma_k \circ \alpha_k(L_n) \subseteq L$$

This implies $L = \gamma_k \circ \alpha_k(L_n)$, meaning that the sequence (5.4) of k-TSS languages converges to L. □

5.3 Learning Unions of k-TSS Languages

In this section, we present guarantees concerning learnability in the limit of unions of k-TSS languages. Then, we present an algorithm merging closest and compatible k-TSS languages.

5.3.1 Generalities



(A) $\gamma_3(Z)$ and $\gamma_3(Z')$. (B) $\gamma_3(Z) \cup \gamma_3(Z')$.

FIGURE 5.1: k-testable languages are not closed under union.

It is well-known that the class of k-testable languages in the strict sense is not closed under union. Take for instance the two 3-testable languages, represented by their DFA in Figure 5.1a, that are generated by the following 3-test vectors:

$$\begin{aligned} Z &= \langle \{aa\}, \{aa\}, \{aaa\}, \{aa\} \rangle \\ Z' &= \langle \{ba, bb\}, \{ab, bb\}, \{baa, bab, aaa, aab\}, \{bb\} \rangle \end{aligned}$$

with $\Sigma = \{a, b\}$. The union $\gamma_3(Z) \cup \gamma_3(Z')$ of these languages, represented by its DFA in Figure 5.1b, is not a 3-testable language. Indeed, it is not a k-testable language for any value of $k > 0$. For $k = 1$, the only k-testable language that extends $\gamma_3(Z) \cup \gamma_3(Z')$ is Σ^*. For $k \geq 2$, the problem is that since a^{k-1} is an allowed prefix, $a^{k-1}b$ is an allowed segment, and $a^{k-2}b$ is an allowed suffix, $a^{k-1}b$ has to be in the language, even though it is not an element of $\gamma_3(Z) \cup \gamma_3(Z')$.

It turns out that we can generalize Theorem 5.2.10 to unions of k-TSS languages.

Theorem 5.3.1. *Any language that is a union of k-TSS languages can be identified in the limit from positive examples.*

Proof. Let $L = L_1 \cup \cdots \cup L_l$, where all the L_p are k-TSS languages, and let $w_1, w_2, w_3, \ldots$ be an enumeration of L. Define, for $i > 0$,

$$K_i = \bigcup_{j=1}^{i} \gamma_k \circ \alpha_k(\{w_j\}).$$

Since each w_j is included in a k-TSS language contained in L, and $\gamma_k \circ \alpha_k(\{w_j\})$ is the smallest k-TSS language that contains w_j, we conclude that, for all j, $\gamma_k \circ \alpha_k(\{w_j\}) \subseteq L$, which in turn implies $K_i \subseteq L$. Since there are only finitely many k-test vectors and finitely many k-TSS languages, the sequence

$$K_1 \subseteq K_2 \subseteq K_3 \subseteq \cdots \tag{5.6}$$

converges, that is there exists an n such that, for all $m \geq n$, $K_m = K_n$. This implies that all w_j are included in K_n, that is $L \subseteq K_n$. In combination with the above observation that all K_i are contained in L, this proves that sequence (5.6) converges to L. □

The proof of Theorem 5.3.1 provides us with a simple first algorithm to learn unions of k-TSS languages: for each example word that we see, we compute the k-test vector, and then we take the union of the languages denoted by all those k-test vectors. The problem with this algorithm is that potentially we end up with a huge number of different k-test vectors. Thus we would like to cluster as many k-test vectors in the union as we can, without changing the overall language. Before we can introduce our clustering algorithm, we first need to define a metric on k-test vectors.

Definition 5.3.2. *The* cardinality *of a k-test vector $Z = \langle I, F, T, C \rangle$ is defined by:*

$$|Z| = |I| + |F| + |T| + |C \cap \Sigma^{<k-1}|.$$

Definition 5.3.3. *The function $d \colon \mathcal{T}_k \times \mathcal{T}_k \mapsto \mathbb{R}^+$, which defines the* distance *between a pair of k-test vectors, is given by: $d(Z, Z') = |Z \triangle Z'|$.*

Intuitively, the distance between two k-test vectors is the number of prefixes, suffixes, substrings and short words that must be added/removed to transform one k-test vector into the other. For examples, see Fig. 5.2b.

Lemma 5.3.4. *Let Z, Z' and Z'' be k test vectors. Then*

$$|Z \triangle Z'| \leq |Z| + |Z'| \tag{5.7}$$
$$Z \triangle \bot = Z \tag{5.8}$$
$$Z \triangle Z' = \bot \Leftrightarrow Z = Z' \tag{5.9}$$
$$Z \triangle Z' = Z' \triangle Z \tag{5.10}$$
$$Z \triangle (Z' \triangle Z'') = (Z \triangle Z') \triangle Z'' \tag{5.11}$$
$$Z \triangle Z' = (Z \triangle Z'') \triangle (Z'' \triangle Z') \tag{5.12}$$

Proof. Let $Z = \langle I, F, T, C\rangle$, $Z' = \langle I', F', T', C'\rangle$ and $Z'' = \langle I'', F'', T'', C''\rangle$. Then equality (5.7) can be derived as follows:

$$\begin{aligned}
|Z \triangle Z'| &= |I \triangle I'| + |F \triangle F'| + |T \triangle T'| \\
&\quad + |(C \triangle C' \triangle (I' \cap F) \triangle (I \cap F')) \cap \Sigma^{<k-1}| \\
&= |I \triangle I'| + |F \triangle F'| + |T \triangle T'| \\
&\quad + |(C \cap \Sigma^{<k-1}) \triangle (C' \cap \Sigma^{<k-1}) \\
&\quad \triangle (I' \cap F \cap \Sigma^{<k-1}) \triangle (I \cap F' \cap \Sigma^{<k-1})| \\
&= |I \triangle I'| + |F \triangle F'| + |T \triangle T'| \\
&\quad + |(C \cap \Sigma^{<k-1}) \triangle (C' \cap \Sigma^{<k-1}) \triangle \emptyset \triangle \emptyset| \\
&= |I \triangle I'| + |F \triangle F'| + |T \triangle T'| \\
&\quad + |(C \cap \Sigma^{<k-1}) \triangle (C' \cap \Sigma^{<k-1})| \\
&\leq |I| + |I'| + |F| + |F'| + |T| + |T'| \\
&\quad + |(C \cap \Sigma^{<k-1})| + |(C' \cap \Sigma^{<k-1})| \\
&= |Z| + |Z'|
\end{aligned}$$

Identities (5.8), (5.9), (5.10) and (5.11) follow from the definitions and the corresponding identities for sets:

$$\begin{aligned}
Z \triangle \bot &= \langle I \triangle \emptyset, F \triangle \emptyset, T \triangle \emptyset, C \triangle \emptyset \triangle (\emptyset \cap F) \triangle (I \cap \emptyset)\rangle \\
&= \langle I, F, T, C\rangle = Z \\
Z \triangle Z' = \bot &\Leftrightarrow I \triangle I' = \emptyset \wedge F \triangle F' = \emptyset \wedge T \triangle T' = \emptyset \\
&\quad \wedge C \triangle C' \triangle (I' \cap F) \triangle (I \cap F') = \emptyset \\
&\Leftrightarrow I = I' \wedge F = F' \wedge T = T' \\
&\quad \wedge C \triangle C' \triangle (I' \cap F) \triangle (I \cap F') = \emptyset \\
&\Leftrightarrow I = I' \wedge F = F' \wedge T = T' \\
&\quad \wedge C \triangle C' \triangle (I \cap F) \triangle (I \cap F) = \emptyset \\
&\Leftrightarrow I = I' \wedge F = F' \wedge T = T' \wedge C \triangle C' \triangle \emptyset = \emptyset \\
&\Leftrightarrow I = I' \wedge F = F' \wedge T = T' \wedge C \triangle C' = \emptyset \\
&\Leftrightarrow I = I' \wedge F = F' \wedge T = T' \wedge C = C' \\
&\Leftrightarrow Z = Z' \\
Z \triangle Z' &= \langle I \triangle I', F \triangle F', T \triangle T', C \triangle C' \triangle (I' \cap F) \triangle (I \cap F')\rangle \\
&= \langle I' \triangle I, F' \triangle F, T' \triangle T, C' \triangle C \triangle (I \cap F') \triangle (I' \cap F)\rangle \\
&= Z' \triangle Z \\
Z \triangle (Z' \triangle Z'') &= \langle I \triangle (I' \triangle I''), F \triangle (F' \triangle F''), T \triangle (T' \triangle T''), \\
&\quad C \triangle (C' \triangle C'' \triangle (I'' \cap F') \triangle (I' \cap F'')) \\
&\quad \triangle((I' \triangle I'') \cap F) \triangle (I \cap (F' \triangle F''))\rangle \\
&= \langle I \triangle (I' \triangle I''), F \triangle (F' \triangle F''), T \triangle (T' \triangle T''), \\
&\quad C \triangle C' \triangle C'' \triangle (I'' \cap F') \triangle (I' \cap F'') \triangle (I' \cap F) \\
&\quad \triangle(I'' \cap F) \triangle (I \cap F') \triangle (I \cap F'')\rangle \\
&= \langle (I \triangle I') \triangle I'', (F \triangle F') \triangle F'', (T \triangle T') \triangle T'', \\
&\quad (C \triangle C' \triangle (I' \cap F) \triangle (I \cap F')) \triangle C'' \\
&\quad \triangle(I'' \cap (F \triangle F')) \triangle ((I \triangle I') \cap F'')\rangle \\
&= (Z \triangle Z') \triangle Z''
\end{aligned}$$

Finally, equality (5.12) follows from the preceding equalities (5.8), (5.9) and (5.11):

$$\begin{aligned}
Z \triangle Z' &= (Z \triangle \bot) \triangle Z' \\
&= (Z \triangle (Z'' \triangle Z'')) \triangle Z' \\
&= ((Z \triangle Z'') \triangle Z'') \triangle Z' \\
&= (Z \triangle Z'') \triangle (Z'' \triangle Z')
\end{aligned}$$

□

Proposition 5.3.5. *Distance function d is a metric.*

Proof. In order to prove that d is a metric, we need to show that it satisfies the following four properties, for all $Z, Z', Z'' \in \mathcal{T}_k$,

1. $d(Z, Z') \geq 0$ (non-negativity)
2. $d(Z, Z') = 0$ iff $Z = Z'$ (identity of indiscernibles)
3. $d(Z, Z') = d(Z', Z)$ (symmetry)
4. $d(Z, Z'') \leq d(Z, Z') + d(Z', Z'')$ (triangle inequality)

Non-negativity follows immediately from the definition of d. Identity of indiscernibles can be derived using Lemma 5.3.4(5.9):

$$d(Z, Z') = 0 \quad \Leftrightarrow \quad |Z \mathbin{\triangle} Z'| = 0 \Leftrightarrow Z \mathbin{\triangle} Z' = \bot \Leftrightarrow Z = Z'.$$

Symmetry follows since $\triangle$ commutes (Lemma 5.3.4(5.10)):

$$d(Z, Z') = |Z \mathbin{\triangle} Z'| = |Z' \mathbin{\triangle} Z| = d(Z', Z).$$

The triangle inequality follows using identities (5.7) and (5.12) from Lemma 5.3.4:

$$\begin{aligned} d(Z, Z') &= |Z \mathbin{\triangle} Z'| \\ &= |(Z \mathbin{\triangle} Z'') \mathbin{\triangle} (Z'' \mathbin{\triangle} Z')| \\ &\leq |Z \mathbin{\triangle} Z''| + |Z'' \mathbin{\triangle} Z'| \\ &= d(Z, Z'') + d(Z'', Z'). \end{aligned}$$

□

The next proposition provides a necessary and sufficient condition for the union of the languages of two k-test vectors to be equal to the language of the union of these k-test vectors.

Proposition 5.3.6. *Suppose $Z = \langle I, F, T, C \rangle$ and $Z' = \langle I', F', T', C' \rangle$ are canonical k-test vectors, for some k. Let $\bullet \notin \Sigma$ be a fresh symbol, and let $G = (V, E)$ be the directed graph with*

$$\begin{aligned} V &= \{\bullet u \mid u \in I \cup I'\} \cup T \cup T' \cup \{u\bullet \mid u \in F \cup F'\}, \\ E &= \{(au, ub) \in V \times V \mid a, b \in \Sigma \cup \{\bullet\}, u \in \Sigma^{k-1}\}. \end{aligned}$$

Suppose each vertex in V is colored either red, blue or white. Vertices in $T \setminus T'$ are red, vertices in $T' \setminus T$ are blue, and vertices in $T \cap T'$ are white. A vertex $\bullet u$ is red if $u \in I \setminus I'$, blue if $u \in I' \setminus I$, and white if $u \in I \cap I'$. A vertex $u\bullet$ is red if $u \in F \setminus F'$, blue if $u \in F' \setminus F$, and white if $u \in F \cap F'$. Then $\gamma_k(Z \sqcup Z') = \gamma_k(Z) \cup \gamma_k(Z')$ iff there exists no path in G from a red vertex to a blue vertex, nor from a blue vertex to a red vertex.

Proof. Let Π be the set of paths in G from a vertex in $\{\bullet u \mid u \in I \cup I'\}$ to a vertex in $\{u\bullet \mid u \in F \cup F'\}$. There exists a 1-to-1 correspondence between paths in Π and strings in $\gamma_k(Z \sqcup Z')$ with length at least $k-1$. Because suppose

$$w = a_1 \cdots a_m$$

is a string in $\gamma_k(Z \sqcup Z')$, for some $m \geq k-1$. Let $a_0 = \bullet$, $a_{m+1} = \bullet$ and let, for $0 \leq j \leq m-k+2$,

$$w_j = a_j \cdots a_{j+k-1}$$

be the substring of $a_0 a_1 \cdots a_m a_{m+1}$ with length k starting at a_j. Using the fact that $w \in \gamma_k(Z \sqcup Z')$, the reader can easily check that all the string w_j are vertices from V. Moreover, for all $0 \leq j < m-k+2$, the pair (w_j, w_{j+1}) is an edge of E. Thus we may conclude that sequence $w_0, \ldots, w_{m-k+2}$ is a path in Π. Conversely, it is trivial to extract from each path in Π a corresponding string in $\gamma_k(Z \sqcup Z')$ with length at least $k-1$. Moreover, paths in Π that only visit red and white vertices correspond with words in $\gamma_k(Z)$, paths in Π that only visit blue and white vertices correspond with words in $\gamma_k(Z')$, and paths in Π that contain both a red and a blue vertex correspond with words in $\gamma_k(Z \sqcup Z') \setminus (\gamma_k(Z) \cup \gamma_k(Z'))$.

$\Rightarrow$ Suppose there exists a path in G from a red vertex v to a blue vertex w. (The case in which there exists a path from a blue vertex to a red vertex is symmetric.) Since Z and Z' are canonical, each red vertex in V occurs on a path in Π that only contains red or white vertices, and each blue vertex in V occurs on a path in Π that only contains blue or white vertices. This means we can construct a path $\pi \in \Pi$ that contains both the red vertex v and the blue vertex w. By the above observations, path π corresponds to a word in $\gamma_k(Z \sqcup Z')$ that is not in $\gamma_k(Z) \cup \gamma_k(Z')$.

$\Leftarrow$ Now suppose there exists no path in G from a red vertex to a blue vertex, nor from a blue vertex to a red vertex. Then any path in Π either contains only red and white vertices, or it contains only blue and white vertices. This implies that any word in $\gamma_k(Z \sqcup Z')$ with length at least $k-1$ is contained either in $\gamma_k(Z)$ or in $\gamma_k(Z')$. By construction all the words in $\gamma_k(Z \sqcup Z')$ with length less than $k-1$ are contained either in $C \subseteq \gamma_k(Z)$ or in $C' \subseteq \gamma_k(Z')$. Thus we may conclude $\gamma_k(Z \sqcup Z') \subseteq \gamma_k(Z) \cup \gamma_k(Z')$. Since γ_k is monotone (Proposition 5.2.6), the converse inclusion $\gamma_k(Z \sqcup Z') \supseteq \gamma_k(Z) \cup \gamma_k(Z')$ holds as well. Thus we may conclude $\gamma_k(Z \sqcup Z') = \gamma_k(Z) \cup \gamma_k(Z')$, as required. □

Suppose alphabet Σ contains n elements. Then the size of graph G is in $O(n \cdot |Z \cup Z'|)$, and we can construct G from Z and Z' in time $O((n+k) \cdot |Z \cup Z'|)$. Since the reachability property in Proposition 5.3.6 can be decided in a time that is linear in the size of G, we obtain an $O((n+k) \cdot |Z \cup Z'|)$-time algorithm for deciding $\gamma_k(Z \sqcup Z') = \gamma_k(Z) \cup \gamma_k(Z')$.

5.3.2 Efficient algorithm

Algorithm 2 Linkage of k-test vectors.

Input: $S = \{\alpha_k(\{x\}) \mid x \in \mathcal{S}\}$: sample of m k-test vectors, D: related distance matrix
Output: L: unsorted dendrogram of k-test vectors.

$chain \leftarrow [\,]$
while $|S| > 1$ ***and*** $\exists a, b \in S$ *s.t.* $\gamma_k(a \sqcup b) = \gamma_k(a) \cup \gamma_k(b)$ **do**
  if $\mathtt{length}(chain) \leq 3$ **then**
    $a \leftarrow$ (any element of S)
    $chain \leftarrow [a]$
    $b \leftarrow$ (any element of $S \setminus \{a\}$)
  end
  else
    $a \leftarrow chain[-4]$
    $b \leftarrow chain[-3]$
    Remove $chain[-1]$, $chain[-2]$ and $chain[-3]$
  end
  repeat
    $c \leftarrow argmin_{x \neq a} d[x, a]$ with preference for b
    $a, b \leftarrow c, a$
    if $\gamma_k(a \sqcup b) = \gamma_k(a) \cup \gamma_k(b)$ `// check if` $\gamma_k(a) \cup \gamma_k(b)$ `is k-TSS`
    then
      Append a to $chain$
    end
  until $\mathtt{length}(chain) \geq 3$ ***and*** $a = chain[-3]$
  Append $(a, b, D[a, b])$ to L
  Remove a, b from S
  $D[a \sqcup b, x] = D[x, a \sqcup b] \leftarrow d(a \sqcup b, x), \forall x \in S$ `// Update D`
  $S \leftarrow S \cup \{a \sqcup b\}$ `// new node is the union of the two k-test vectors`
end
return L

Our algorithm to learn unions of k-testable languages is based on hierarchical clustering. Given a set $\mathcal{S}$ of n words, we compute its related set of k-test vectors $S = \{\alpha_k(\{x\}) \mid x \in \mathcal{S}\}$. Note that the k-test vectors are canonical. Then, an $n \times n$ distance matrix is computed. To that end, the distance used is the pairwise distance between k-test vectors defined in Definition 5.3.3. Next, the algorithm finds the closest pair of compatible k-test vectors Z and Z', such that $\gamma_k(Z \sqcup Z') = \gamma_k(Z) \cup \gamma_k(Z')$ and computes their union. The distance between the merged k-test vectors and the remaining k-test vectors in S is updated. These two operations are repeated until all initial k-test vectors have been merged into one, or that no allowed union of two k-test vectors such that $\gamma_k(Z \sqcup Z') = \gamma_k(Z) \cup \gamma_k(Z')$ is possible. We gather at the end of the process a linkage between k-test vectors, which can lead to the computation of a dendrogram. When the number of k-test vectors to learn is known, one can use this expected number of languages to find the threshold that would, given the hierarchical clustering, return the desired unions of k-test vectors.

An efficient implementation for such hierarchical clustering algorithms is the nearest-neighbour chain algorithm [15]. Our hierarchical clustering of k-test vectors is presented in Algorithm 2. We reuse notations from [103].

Example 5.3.7. *Let $k = 3$. Given the sample of strings $\mathcal{S}$ in Table 5.2a, compute the associate sample of 3-test vectors $S = \{Z_1, Z_2, \ldots, Z_8\}$. Then, compute its distance matrix (Table 5.2b) using the metric defined in Definition 5.3.3. Using classical linkage algorithms (for instance nearest-neighbour chain algorithm), compute the related linkage matrix depicted in Table 5.2c. We gather the dendrogram shown in Figure 5.2d, where the 3 remaining 3-test vectors $Z_1 \sqcup Z_8$, $Z_2 \sqcup Z_5 \sqcup Z_7$ and $Z_3 \sqcup Z_4 \sqcup Z_6$ cannot be merged. Indeed:*

- $\gamma_k(Z_1 \sqcup Z_8 \sqcup Z_2 \sqcup Z_5 \sqcup Z_7) \neq \gamma_k(Z_1 \sqcup Z_8) \cup \gamma_k(Z_2 \sqcup Z_5 \sqcup Z_7)$.
- $\gamma_k(Z_1 \sqcup Z_8 \sqcup Z_3 \sqcup Z_4 \sqcup Z_6) \neq \gamma_k(Z_1 \sqcup Z_8) \cup \gamma_k(Z_3 \sqcup Z_4 \sqcup Z_6)$.
- $\gamma_k(Z_2 \sqcup Z_5 \sqcup Z_7 \sqcup Z_3 \sqcup Z_4 \sqcup Z_6) \neq \gamma_k(Z_2 \sqcup Z_5 \sqcup Z_7) \cup \gamma_k(Z_3 \sqcup Z_4 \sqcup Z_6)$.

With a desired number of 3-TSS languages to learn of 3, the returned languages are $\gamma_k(Z_1 \sqcup Z_8)$ and $\gamma_k(Z_2 \sqcup Z_5 \sqcup Z_7)$ and $\gamma_k(Z_3 \sqcup Z_4 \sqcup Z_6)$. With a desired number of 3-TSS languages to learn of 4, the returned languages would be $\gamma_k(Z_1 \sqcup Z_8)$ and $\gamma_k(Z_2 \sqcup Z_5 \sqcup Z_7)$ and $\gamma_k(Z_3)$ and $\gamma_k(Z_4 \sqcup Z_6)$ instead.

$\mathcal{S}$	S
baba	$Z_1 = \langle\{ba\},\{ba\},\{bab,aba\},\{\}\rangle$
abba	$Z_2 = \langle\{ab\},\{ba\},\{abb,bba\},\{\}\rangle$
abcabc	$Z_3 = \langle\{ab\},\{bc\},\{abc,bca,cab\},\{\}\rangle$
cbacba	$Z_4 = \langle\{cb\},\{ba\},\{cba,bac,acb\},\{\}\rangle$
abbbba	$Z_5 = \langle\{ab\},\{ab\},\{abb,bbb,bba\},\{\}\rangle$
cbacbacba	$Z_6 = \langle\{cb\},\{ba\},\{cba,bac,acb\},\{\}\rangle$
abbba	$Z_7 = \langle\{ab\},\{ba\},\{abb,bbb,bba\},\{\}\rangle$
bababc	$Z_8 = \langle\{ba\},\{bc\},\{bab,aba,abc\},\{\}\rangle$

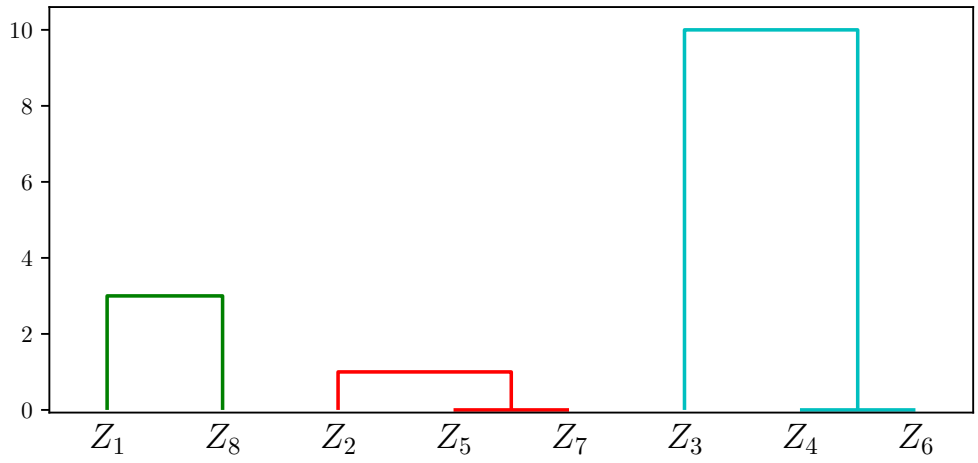
(A) Dataset and corresponding 3-test vectors.

	Z_1	Z_2	Z_3	Z_4	Z_5	Z_6	Z_7	Z_8
Z_1	0	6	9	7	7	7	7	3
Z_2	6	0	7	7	1	7	1	9
Z_3	9	7	0	10	8	10	8	6
Z_4	7	7	10	0	8	0	8	10
Z_5	7	1	8	8	0	8	0	10
Z_6	7	7	10	0	8	0	8	10
Z_7	7	1	8	8	0	8	0	10
Z_8	3	9	6	10	10	10	10	0

(B) Distance matrix.

Z_5	Z_7	0
Z_4	Z_6	0
Z_2	$Z_5 \sqcup Z_7$	1
Z_1	Z_8	3
Z_3	$Z_4 \sqcup Z_6$	10

(C) Linkage matrix.



(D) Corresponding dendrogram.

FIGURE 5.2: Learning unions of k-test vectors.

Proposition 5.3.8. *The time complexity of learning union of k-test vectors using Algorithm 2 is in* $\mathcal{O}(mk \cdot |x| + 2^{n^k} \cdot m^2 + (n+k) \cdot 2^{n^k} \cdot m^2)$.

Proof. Suppose alphabet Σ contains n elements and that the sample contains m elements. Let x be the longest word in the sample. The learning of unions of k-test vectors is done in 3 steps:

- Computing the initial k-test vectors, which is in $O(mk \cdot |x|)$ with x the longest word in the sample.
- Building the distance matrix is in $O(2^{n^k} \cdot m^2)$.
- Building the hierarchical clustering (linkage of neighbouring k-test vectors) is in $O((n+k) \cdot 2^{n^k} \cdot m^2)$.

□

5.4 Case Study

Job dataset Our case study has been inspired by an industrial problem related to the domain of cyber-physical systems. Recent work [141] focused on the impact of design parameters of a flexible manufacturing system on its productivity. It appeared in the aforementioned study that productivity depends on the jobs being rendered. To that end, the prior identification of the different job patterns is crucial in enabling engineers to optimize parameters related to the flexible manufacturing system.

TABLE 5.1: Sample of identified job patterns.

job	pattern	3-test vector	type of job
aaaaa	a^+	$Z = \langle\{aa\}, \{aa\}, \{aaa\}, \{a, aa\}\rangle$	homogeneous
aaaaaaaaaa			
aaaaa . . . aaa			
abababab	$(ab)^+$	$Z = \langle\{ab\}, \{ab\}, \{aba, bab\}, \{ab\}\rangle$	heterogeneous
abababababab			
abcabcabc	$(abc)^+$	$Z = \langle\{ab\}, \{bc\}, \{abc, bca, cab\}, \{\}\rangle$	
abcabcabcabcabc			
abcbcbcbca	$a(bc)^+a$	$Z = \langle\{ab\}, \{ca\}, \{abc, bcb, cbc, cba\}, \{\}\rangle$	miscellaneous

We consider a dataset containing strings, each representing a job. Our job patterns are also represented by 3-testable languages, the 3-test vectors of which are shown in Table 5.1.

Our dataset, implementations and complete results are available[1].

[1] https://gitlab.science.ru.nl/alinard/learning-union-ktss

5.5 Conclusion

In this chapter, we defined a Galois connection characterizing k-testable languages. We also described an efficient algorithm to learn unions of k-testable languages that results from this Galois connection. From a practical perspective, we see that obtaining more than one representation is meaningful since a too generalized solution is not necessarily the best. To avoid unnecessary generalizations, the union of two k-testable languages that would not be a k-testable language is not allowed. Note also that depending on the applications, expert knowledge can indicate the number of languages the returned union should contain. In further work, we would like to extend the learning of unions of languages to regular languages. An attempt to learn pairwise disjoint regular languages has been made in [91, 94]. However, no learnability guarantee has been provided so far. This is further discussed in Chapter 6. From a practical perspective, we refer to Chapter 7, where the result of our method is applied to a Large Scale Printer. We show in this chapter that information on print job patterns (described by the union of languages learned from a set of traces describing the print jobs) improves the productivity of the printers for some of the print jobs that are rendered.

Chapter 6

Towards Learning Unions of Regular Languages

A classical problem in grammatical inference is to identify a Deterministic Finite Automaton (DFA) from a set of positive and negative examples. In this chapter, we address the problem of identifying a set of DFA from examples that belong to several *different* unknown regular languages. We propose two methods for learning from positive examples, based on compression for clustering the observed positive examples. We also propose two methods to learn from positive and negative examples, one based on state merging and the other on a multi-objective evolutionary algorithm. We evaluate our methods on an industrial dataset.

6.1 Introduction

A common problem in grammatical inference is to find, i.e. *learn*, a regular language from a set of examples of that language. When this set is divided into positive examples (belonging to the language) and negative examples (not belonging to the language), the problem is typically solved by searching for a DFA that accepts the positive examples and rejects the negative ones. Provided with sufficiently many examples, there exist algorithms that will correctly learn an unknown language from these examples [63].

We consider a setting where one can observe positive examples from multiple different regular languages, but it is not clear to which of these languages the examples belong. Given the following set of strings $S = \{aa, aaa, aaaa, abab, ababab, abba, abbba, abbbba\}$, gathering a single automaton might be less informative than gathering several DFA respectively encoding the languages a^*, $(ab)^*$ and ab^*a. There is a trade-off between the number of languages and how specific each language should be. That is, covering all words through a single language may not be the desired result, but having a language for each word may also not be desired. Moreover, we assume that each example belongs to exactly one of these languages, i.e. that the languages are disjoint. As a result, the positive examples for one language are negative examples for the other languages. The problem at hand is to cluster the examples and learn the corresponding languages.

Unlike k-testable languages presented in Chapter 5, regular languages are not learnable in the limit from positive examples only. Similarly, their union is neither

learnable from positive examples, nor from positive *and* negative examples (see Sect. 6.2). The focus of this chapter is, therefore, on providing algorithms that would heuristically find interesting unions, rather than a theoretical study on the non-learnability of unions of regular languages.

Considering learning from text (positive examples only), we present two clustering approaches based on an essential property of regular languages described by the pumping lemma for regular languages [72]. The pumping lemma is often used to prove that a particular language is *not* regular. It states that any sufficiently long example from a regular language has a middle section that may be repeated (i.e. *pumped*) to produce a new example that is also in that language [119]. Efficient algorithms exist for finding such decompositions [136]. We use these possible decompositions for clustering our examples. The intuition behind this approach is that examples from the same language are more likely to have observable similarities than examples from different languages. Based on this assumption, clustering by compression [41] can also be used to regroup the strings.

Considering a setting where one learns from a complete presentation (positive and negative examples, which are strings belonging to none of the languages to learn), we propose a modification of the state merging algorithm RPNI [111], to output a set of DFA. We also propose an evolutionary algorithm inspired by [52] whose fitness function takes into account the accuracy of the DFA w.r.t. the learning sample – that is to say, ensuring that all positive strings are accepted and all negative strings rejected – while also taking into account the desired number of DFA.

Our problem is motivated by a case study of print jobs submitted to large industrial printers. Strings of symbols represent these print jobs, where each symbol denotes a different media type, such as a book cover or a newspaper page. Together, this set of print jobs makes for a reasonably complicated 'language'. Nevertheless, we observed that each print job could be classified as belonging to one of a fixed set of categories, such as 'book' or 'newspaper'. Two print jobs that belong to the same category are typically very similar, to the extent that they only differ in the number of repetitions of a particular media type. Therefore, the languages stand for the different families of print jobs. Our goal is to uncover these simple languages, which are found to be star-free. The idea of decomposing a regular language into smaller pieces which are easier to represent has been suggested in [61] in the context of automated XML style sheet extraction, as well as in [60] in the context of learning teams of automata. However, none of them considered the task of heuristically learning unions of languages from data.

This chapter is organized as follows. In Section 6.2 we recall preliminary definitions. We present then in Sections 6.3 to 6.6 our techniques to infer several languages from text or complete presentation. In Section 6.7, we show the results we achieved on an industrial case study. Finally, we discuss and conclude about the comparison of the proposed methods.

6.2 Definitions

In this section, we present the different concepts and definitions related to the learning of a set of DFA from strings.

Strings Let Σ denote a finite alphabet of *symbols*. A *string* $x = a_0 \dots a_n$ is a finite sequence of symbols. The empty string is denoted ϵ. We denote by Σ^* the set of all strings over Σ, and by Σ^+ all nonempty strings over Σ (i.e. $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$). Similarly, we denote by Σ^i the set of all strings over Σ of length i. Given a set of strings S, we denote by $\texttt{prefix}(S)$ the set of prefixes of strings in S.

Labeled examples A set of *labeled strings* is a finite set of strings divided into positive strings (belonging to the language to learn) and negative strings (not belonging to the language to learn). We refer to S_+ and S_- as the set of positive and negative examples.

Languages A regular language can be described by a *deterministic finite automaton* (DFA), which is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$, where Q is a finite set of *states*, $q_0 \in Q$ is the *starting state*, $\delta : Q \times \Sigma \to Q$ is a partial *transition function* from states and symbols to states, and $F \subseteq Q$ is a set of *accepting states*. Note that the transition function can also be represented as a *transition matrix* M, being a two-dimensional array of size $|Q| \times |\Sigma|$ where $M_{i,j} = \delta(q_i, a_j)$ with q_i and $M_{i,j} \in Q$ and $a_j \in \Sigma$; and the set of accepting states as an *output array* O of size $|Q|$ where the i-th value of the array is 1 if q_i is accepting, 0 otherwise. Let q be a state, a be a symbol and u be a string. Then we extend δ to $\delta^* : Q \times \Sigma^* \to Q$ by $\delta^*(q, \epsilon) = q$ and $\delta^*(q, au) = \delta^*(\delta(q, a), u)$. The set of tails described by a state q is the set $\{x \in \Sigma^* : \delta^*(q, x) \in F\}$. Hence, the language $\mathcal{L}(\mathcal{A})$ recognized by DFA $\mathcal{A}$ is the set of tails of the starting state of $\mathcal{A}$. The Prefix Tree Acceptor (PTA) of a non empty set S_+ is a tree-like DFA $\mathcal{A} = (\Sigma, Q, \epsilon, \delta, S_+)$ such that $Q = \texttt{prefix}(S_+)$ and $\forall ua \in \texttt{prefix}(S_+) : \delta(u, a) = ua$. A language is said to be *star-free* if it can be defined using a regular expression without Kleene star (but possibly with the complement operator). It has been shown that a language $\mathcal{L}$ is star-free if its syntactic monoid is finite and aperiodic [130]. It has also been shown in [99] that if a language $\mathcal{L}$ is star-free, then the minimal automaton for $\mathcal{L}$ is counter-free. This means, $\exists i, j \geq 1$ such that $\forall u, v, w \in \Sigma^*$ and $j \neq i$ we have $uv^j w \in \mathcal{L} \Leftrightarrow uv^i w \in \mathcal{L}$.

Learnability in the limit The concept of learnability in the limit has been introduced in [63]. A class of languages is said to be *learnable in the limit* if, after reading a given number of strings (labelled or not), a learning algorithm will converge to the correct target language in a finite number of iterations. It has been demonstrated that regular languages are learnable in the limit from a complete presentation (i.e. from labelled examples), but not from text (i.e. positive examples only). The interpretation is the following: given only strings which are known to be in the languages, no algorithm can decide whether to generalize to a

coarse language (i.e. tending to Σ^*) or to a more specific language (i.e. accepting only the strings seen in the dataset). Making such decisions would be either arbitrary, or either depending on heuristics. Interestingly, we can extend this reasoning to unions of regular languages. We claim that the union of regular languages is neither learnable in the limit from text nor from a complete presentation: indeed, deciding whether this or that string should be in one of the languages in the union rather than in another seems, again, arbitrary. Hence, our focus on finding relevant heuristics for learning unions of languages in this chapter.

Pumping lemma The pumping lemma [119] states that all sufficiently long strings of a regular language contain an infix that may be *pumped*, i.e. repeated an arbitrary number of times to construct a new string that is in the language. Let $\mathcal{L}$ be a regular language, then there exists an integer $p \geq 1$ depending only on $\mathcal{L}$ such that every string $x \in \mathcal{L}$ of length at least p can be written as $x = uvw$ such that $|v| \geq 1$, $|uv| \leq p$ and $\forall i \geq 0$, $uv^iw \in \mathcal{L}$. The intuition is that v will always lead back to the same state, i.e. $\delta^*(q_0, u) = \delta^*(\delta^*(q_0, u), v) = q$. Therefore, a string uv^iw is in the language if and only if w is in the set of tails of q. We say that two strings x and y have the same *pumping decomposition* if $x = uv^iw$ and $y = uv^jw$ for some u, v, w, i and j.

Compression Compression refers to the practice of encoding information using fewer units of information (e.g. *bits*) than the original representation. The function that provides such an encoding is called a *compressor*. We formally define a compressor as a so-called 'code word length function' $C : \Sigma^* \to \mathbb{N}$ that maps strings to the length of their compressed encoding. A normal compressor can be used to approximate the *Kolmogorov complexity* [101] of a string (i.e. the length of the shortest computer program that produces the string as output), as the Kolmogorov complexity itself is not computable. In Cilibrasi and Vitányi [41, Def. 3.1 and Lemma 3.4], a *normal compressor*, e.g. *zip*, is defined as:

Definition 6.2.1. *A compressor C is* normal *if it satisfies, up to an additive $O(log\ n)$ term, with n the maximal length of an element involved in the (in)equality concerned, the following:*

1. $C(uu) = C(u)$ *and* $C(\epsilon) = 0$ *(idempotency)*
2. $C(uv) = C(vu)$ *(symmetry)*
3. $C(uv) \leq C(u) + C(v)$ *(subadditivity)*

The *normalized compression distance* (NCD) is a quasi-universal measure for the similarity of two strings. It is an approximation of their (uncomputable) *information distance*, which measures the minimum information required to go from one string to the other or vice versa. The NCD for two strings x, y is defined in [41] as follows.

$$NCD(x, y) = \frac{max\{C(xy) - C(x), C(yx) - C(y)\}}{max\{C(x), C(y)\}} \tag{6.1}$$

6.3 Clustering by Compression

The NCD can be used to cluster strings based on their pumping decomposition. We propose here a method clustering the strings based on their compression and relate such clustering to the pumping lemma for regular languages. We then use the clustered strings to infer as many languages as clusters found.

Clustering based on NCD Let us consider a setting where $\mathcal{L}$ and $\mathcal{L}'$ are two disjoint star-free languages, i.e. $\mathcal{L} \cap \mathcal{L}' = \emptyset$. Let $x, y \in \mathcal{L}$ two strings such that $x = uv^i w$ and $y = uv^j w$ with $i, j \geq 0$ and $i \neq j$. Let now $z \in \mathcal{L}'$ be a string with $z = rs^k t$, $k \geq 0$ and ($r \neq u$ or $s \neq v$ or $t \neq w$). We now consider a normal compressor C, which would compress any string in $\mathcal{L}$ or $\mathcal{L}'$. Thus, we can say that, up to an additive $O(log\ n)$ term, $\forall x, y \in \mathcal{L} \quad C(x) = C(y)$. We also assume that the normal compressor we use satisfies up to an additive $O(log\ n)$ term the equality $C(xy) = C(x)$, since a normal compressor C would compress equally the concatenation of two similar strings, i.e. belonging to the same language, and the strings separately. Since we know that $C(x) = C(y)$, and that $C(xy) = C(yx)$ by symmetry, we can then reformulate the NCD between x and y as

$$NCD(x, y) = \frac{C(xy) - C(x)}{C(x)} = 0$$

As in our setting, $\mathcal{L}$ and $\mathcal{L}'$ are two disjoint languages, s.t. strings in $\mathcal{L}$ and $\mathcal{L}'$ follow the construction aforementioned, we can thus assume that

$$NCD(x, y) \leq NCD(x, z) \quad \text{and} \quad NCD(x, y) \leq NCD(y, z).$$

By considering pairwise disjoint star-free languages, we can say that a clustering based on the NCD will regroup sufficiently long strings in the format defined above and belonging to the same language. In practice, note that the additive $O(log\ n)$ term may have an impact on the purity of the clustering. We will, therefore, carry out different clusterings of strings, possibly corresponding to clear distinct languages, depending on the granularity of the clustering, and of the number of clusters to find.

Given a set of strings, the pairwise NCDs are used to compute a distance matrix. The next step is to establish clusters from the previously computed distance matrix. The output of hierarchical clustering is a dendrogram, with one leaf for every string. A dendrogram is composed of internal nodes, linking leaves or subtrees. To build the dendrogram, hierarchical agglomerative clustering algorithms find and merge the pair of clusters with the lowest linkage, and create a parent node at the level of the linkage. We gather the dendrogram, and clusters are formed by varying the level of linkage as a threshold. We assume that the number of languages to get, thus the right threshold to set, is known beforehand.

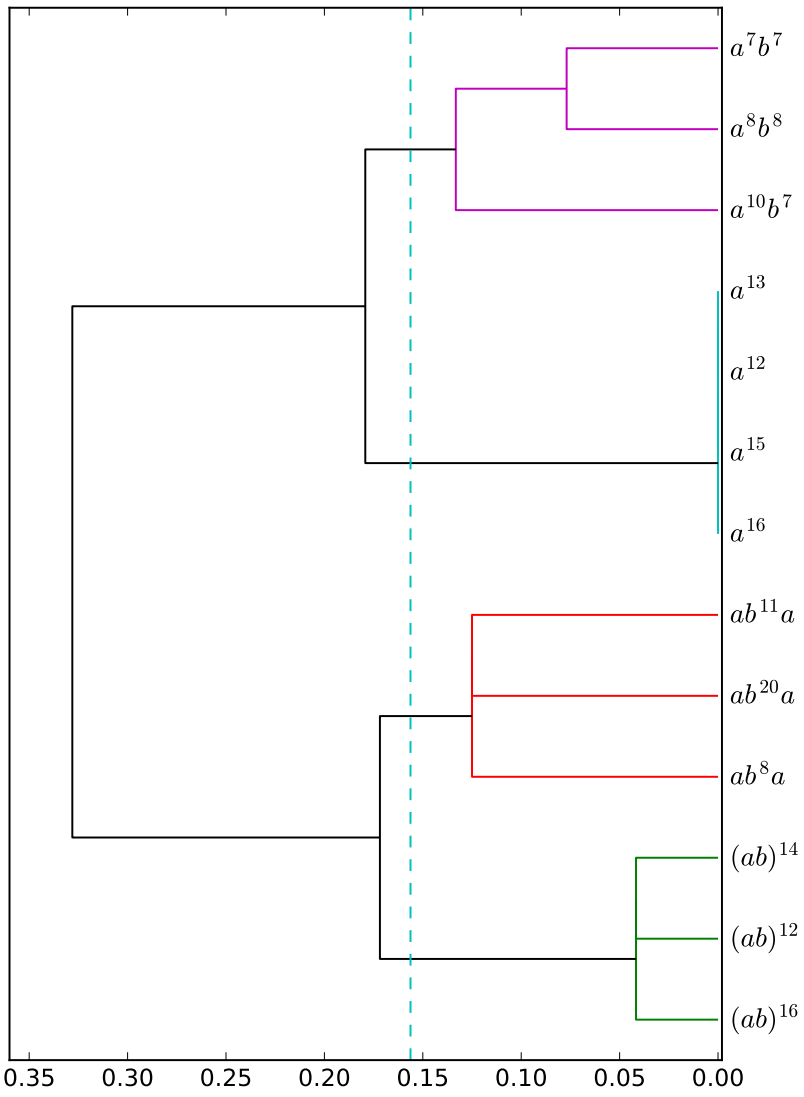


FIGURE 6.1: Clustering of a sample of the printers dataset.

As shown in the example in Figure 6.1, clusters are created and highlighted with different colours, assuming a threshold of 0.16 for the separation. In this case, we gather four sets of strings from which we infer distinct languages. For a given cluster $\mathcal{C}_x$, we can infer a language (for example, by using state merging [70]), using all strings clustered in $\mathcal{C}_x$ as positive examples , and all strings in $S_+ \setminus \mathcal{C}_x$ as negative examples. This process yields for the example 4 languages respectively standing for a^+ (blue), a^+b^+ (purple), ab^+a (red) and $(ab)^+$ (green).

Granularity of Clustering As shown above, choosing the right threshold enables getting the desired number of clusters. Thus, we assume that the right number of clusters is provided in order to enhance the identification of different languages. The granularity of clustering, that is to say, how fine the strings are clustered together, changes depending on this threshold. For example, a solution with a threshold of 0.45 would return three languages, namely ab^+a, a^+b^+ and $((ab)^+|a^+)$ whereas a smaller threshold would provide finer languages (up to singleton sets).

6.4 Tandem Repeats

Another approach is to first compute the pumping decompositions of the strings (and regroup the strings accordingly) by computing *tandem repeats*. We show here that we can infer languages based on the identification of such repetitions in the strings.

Definition 6.4.1. *A string $v \neq \epsilon$ is a* tandem repeat *in x if $x = uv^iw$ for some $i \geq 2$, $u, w \in \Sigma^*$.*

There exist linear time algorithms that can detect all primitive tandem repeats (i.e. not containing shorter tandem repeats) in a string [7, 45, 66, 136]. We relate the pumping lemma for regular languages to the identification of tandem repeats, in a context of star-free languages.

Identification of languages through pumping decompositions It has been shown in [136] that the number of tandem repeats in a string of length x is bounded by $O(x \log x)$. Here, we base our identification of star-free languages on the assumption that two sufficiently long strings belong to the same language if they contain a similar prefix, tandem repeat, and suffix.

Algorithms for finding tandem repeats in a string can thus be used to cluster strings. Let us consider the string $x = uvvw$. Tandem repeat finders will naturally find the substring v as being a tandem repeat. As a consequence, we can use the tandem repeat found to describe the string as $x = uv^2w$. We can then generalize this to infer the language $\{uv^iw : i \geq 2\}$.

Now, let us consider a string $x = abcbcbcbcd$. String x contains three distinct tandem repeats, namely bc, cb and $bcbc$. By taking the longest tandem repeat occurring the most, that is not a tandem repeat itself, we can say $x =$

$a(bc)^4d$. By generalizing, we can infer the language $\{a(bc)^id : i \geq 2\}$. Tandem repeats can also be computed recursively. Let us consider the string $x = aaabcbcaaabcbcaaabcbc$. We can first detect the tandem repeat $aaabcbc$, and state that $x = (aaabcbc)^3$. If we again process the result, we can detect tandem repeats a and bc, and describe x as $(a^3(bc)^2)^4$. By generalizing, we can infer the language $\{(a^i(bc)^j)^k : i, j, k \geq 2\}$.

Ambiguities Let x be the string $abaabaab$. Then, x contains several possible tandem repeats, namely $t_1 = baa$, $t_2 = aab$ or $t_3 = aba$. Thus, we can describe the language for x in different ways, namely $\mathcal{L}_1 = a(baa)^{\geq 2}b$, $\mathcal{L}_2 = ab(aab)^{\geq 2}$ or $\mathcal{L}_3 = (aba)^{\geq 2}ab$. Nevertheless, all these representations are equal, i.e. $\forall i \geq 0$, $a(baa)^ib = ab(aab)^i = (aba)^iab$. Therefore, we claim that any of these representations are valid in order to infer the language. It does not matter which one we choose, as long as we check for equivalence.

<table>
<tr><td>abc</td><td colspan="3">abc</td><td rowspan="2">$(abc)^*$</td></tr>
<tr><td>$abcabc$</td><td colspan="2">$(abc)^{\geq 2}$</td><td>$(abc)^+$</td></tr>
<tr><td>$ababab$</td><td colspan="3">$(ab)^{\geq 2}$</td><td rowspan="2">$(ab)^+$</td></tr>
<tr><td>$abababab$</td><td colspan="3"></td></tr>
<tr><td>$aabaabaab$</td><td rowspan="2">$(aab)^{\geq 2}$</td><td rowspan="2">$(a^{\geq 2}b)^{\geq 2}$</td><td rowspan="2">$(a^+b)^+$</td><td rowspan="3">Σ^*</td></tr>
<tr><td>$aabaabaabaabaab$</td></tr>
<tr><td>$aabbbaabbbaabbb$</td><td>$(aabbb)^{\geq 2}$</td><td rowspan="2">$(a^{\geq 2}b^{\geq 2})^{\geq 2}$</td><td rowspan="2">$(a^+b^+)^+$</td></tr>
<tr><td>$aaabbaaabbaaabb$</td><td>$(aaabb)^{\geq 2}$</td></tr>
<tr><td>$aaabbaaabbaabbbaabbb$</td><td>$(aaabb)^{\geq 2}(aabbb)^{\geq 2}$</td><td>$(a^{\geq 2}b^{\geq 2})^{\geq 2}(a^{\geq 2}b^{\geq 2})^{\geq 2}$</td><td></td></tr>
</table>

TABLE 6.1: Clustering of languages, from fine grained solutions (left) to coarse (right).

Hierarchical clustering of languages We have defined several steps to generalize pumping decompositions of strings based on tandem repeats. As shown above, we can describe strings as $x = uv^iw$ for all $i \geq 2$, and cluster them into $uv^{\geq 2}w$. We then propose to lower the multiplicity of v to ≥ 1, to get uv^+w. The next step of generalization is to get uv^*w by lowering the multiplicity of v to ≥ 0. We can then imagine Σ^* to be an ultimate generalization. Indeed, $\{uvvw\} \subset uv^{\geq 2}w \subset uv^+w \subset uv^*w \subset \Sigma^*$. In this generalization process, some strings might share some generalizations from given steps. In Table 6.1, we illustrate such a hierarchical clustering of regular languages, from singleton sets to Σ^*.

6.5 RPNI for Unions of Languages

We consider now the setting of learning from positive and negative examples, and more specifically, state merging. In RPNI, a PTA is built from the learning sample. Then, the task is to try to generalize this prefix tree by merging states, always ensuring that the resulting automaton is consistent with both negative and positive samples. At any moment of the iterative process of state merging, states are categorized, i.e. marked as either *red* (states that have been analyzed/merged, and that will be part of the final automaton) or *blue* (candidate states, that need to be analyzed to see if merging with a *red* state is possible). RPNI can be seen as based on this *"red/blue"* framework [70]. While performing state merging, it appears that some merges are involving either a considerable number of states (relative to the overall size of the DFA at the current iteration), or a considerable number of accepting states (in comparison with the size of S_+).

The approach assumes the following: in case a merging operation involves *many* states, that is to say, that the size of the DFA drastically changes, then the part being merged has a remarkably distinct meaning with regards to the rest of the DFA. That is, in case a significant number of accepting states collapse all at once, it might be interesting to look into the related strings belonging to S_+ by deriving a separate solution from this sub-sample. We say that we *split* the DFA at this iteration, to get the union of two distinct DFA. The notion of *big merge* can be defined as a function of an input *splitting* parameter k, the larger the more sub-DFA would be returned.

In Algorithm 3, we present a modification of RPNI to output a set of DFA. We follow the notations of [70]. The definitions for `buildPTA`, `rpniCompatible`, `merge` and `promote` are extensively described in the literature (see [70]). The function `buildPTA` returns a PTA of a non empty set S_+. The function `rpniCompatible` returns a boolean indicating whether a DFA is consistent with a labelled dataset. The function `merge` returns a DFA where two specified states have been merged, and adjacent transitions updated. The function `promote` turns a blue state into a red state. Finally, the function `choose` returns the first blue state in alphabetical order. We refer to RPNI without splitting as `standardRPNI`. $|Q_m|$ stands for the number of accepting states being merged during an iteration of state merging, $|S_+|$ the size of the positive sample, $|Q| - |Q_m|$ for the difference of size of the DFA before and after an iteration of state merging. As soon as a consequent merge is detected, we extract all the strings $s \in F \backslash F_m$ to derive a fine-grained subsidiary solution. The idea is to build a sub-PTA accepting all strings $s \in F \backslash F_m$ and rejecting S_-. Note here all the DFA reject S_-. Then, 1. usual state merging is executed, to gather a proper sub-DFA from the particular set of strings originated from the *big merge* extraction and 2. RPNI-union is recursively called with $S_+ \backslash F_m$ as a positive sample and $S_- \cup F_m$ as a negative sample, which means using the strings in S_+ which have not been merged as a positive sample, and adding the merged ones to the negative dataset. At the end of the overall process, we gather as many sub-DFA as detected *big merges*. The main advantage of the application of this heuristic is its genericity: it applies to other state merging algorithms such

Algorithm 3 RPNI-union.

Input: S_+, S_-: set of labeled strings, k: splitting parameter
Output: set of at most k DFA accepting all strings $s \in S_+$ and rejecting all strings $r \in S_-$

$\mathcal{A} \leftarrow \texttt{buildPTA}(S_+)$
$\mathcal{R} \leftarrow \{q_\epsilon\}$, $\mathcal{B} \leftarrow \{q_a : a \in \Sigma \cap \texttt{prefix}(S_+)\}$ `// set of red/blue states`
while $\mathcal{B} \neq \emptyset$ **do**
  $\texttt{choose}(q_b \in \mathcal{B})$
  $\mathcal{B} \leftarrow \mathcal{B} \backslash \{q_b\}$
  if $\exists q_r \in \mathcal{R}$ ***s.t.*** $\mathit{rpniCompatible}(\mathit{merge}(\mathcal{A}, q_r, q_b), S_-)$ **then**
    $\mathcal{A}_m \leftarrow \texttt{merge}(\mathcal{A}, q_r, q_b)$ // $\mathcal{A}_m = (\Sigma, Q_m, q_0, \delta_m, F_m)$
    if $k > 1$ ***and*** $(|F \backslash F_m| \geq \frac{|S_+|}{k}$ ***or*** $|Q| - |Q_m| \geq \frac{|Q|}{k})$ **then**
      return $\{\texttt{standardRPNI}(F \backslash F_m, S_-)\} \cup \texttt{RPNI-union}(S_+ \backslash F_m, S_- \cup F_m, \frac{k}{2})$
    end
    $\mathcal{A} \leftarrow \mathcal{A}_m$
    $\mathcal{B} \leftarrow \mathcal{B} \cup \{\delta(q, a) \in Q \backslash \mathcal{R} : q \in \mathcal{R} \wedge a \in \Sigma\}$
  end
  else
    $\mathcal{R}$, $\mathcal{B} \leftarrow \texttt{promote}(q_b, \mathcal{R}, \mathcal{B}, \mathcal{A})$
  end
end
return $\{\mathcal{A}\}$

as EDSM [81]. Note, however, that disjointness of the returned languages cannot be guaranteed: how state merging is processed within the separate solutions is not controlled to ensure their disjointness. A way to solve this would be to incrementally compute the pairwise intersection between the languages learned so far and to subtract this result to the newly computed sub-DFA.

6.6 Evolutionary Algorithm for Learning Unions of DFA

Evolutionary Algorithms for DFA learning have been investigated in [52] for the first time and consist of the following steps: 1. the generation of an initial population containing a single randomly generated DFA; 2. the evaluation of the fitness of each DFA in the population. Often, the accuracy of the DFA w.r.t. S_+ and S_- is a measure of choice for fitness; 3. performing genetic operations on the individuals; 4. the selection of the best individuals in the population. Steps 2, 3 and four are repeated until at least one individual in the population achieves perfect fitness or a maximum number of allowed iterations (stopping criterion) is reached: then, the best DFA in the population is returned.

Whenever more than one objective has to be optimized (accuracy, size of the DFA), multi-objective evolutionary algorithms such as NSGA-II [47] can be used. They optimize each objective by allowing non-dominated solutions to continue to the next generation. A solution is dominated when another solution is better in every objective. Multi-objective optimization for DFA learning has already been used by [147] in an active learning setting. In our case, we propose to base the selection not only on the accuracy of the DFA but also on the number of desired DFA k. Hence the multi-objectiveness of our algorithm and the use of NSGA-II in our case. We do so by varying k and optimizing for the accuracy while giving a penalty for having a too small or too large amount of languages.

Initial Population The initial operation is to define the seed population. We chose to start with an initial population consisting of one DFA $\mathcal{A}$ per string in S_+ such that $\forall s \in S_+$, $\mathcal{A} = \texttt{buildPTA}(s)$.

Genetic Operators There are two main genetic operators that, given parent(s), generate new individuals using the parent(s) genome as a basis. For a given a parent DFA p_1, mutation consists of randomly mutating an entry in the transition matrix M_{p_1} to a random value in Q. The same operation can be applied to the output array O. The crossover consists of taking parents p_1 and p_2, mixing their genomes, and producing children solutions out of them. Concretely, a random index (single-point) is chosen in the transition matrices M_{p_1}, M_{p_2} of the parents from which top rows from p_1 and the bottom rows of p_2 create one child, and the combination of the other rows another child.

Fitness Functions After a new generation is created, two *scores* are assigned to each individual: f_1 based on the accuracy a of the DFA (proportion of strings in S_+ and S_- which are correctly classified) and f_2 based on the number of sub-DFA n that are calculated out of the individual and given S_+.

$$f_1 = 1 - a \qquad f_2 = |1 - \frac{n}{k}|$$

Transitions Clustering The number of sub-DFA that can be rendered out of a single DFA is calculated by looking at the number of different paths taken by the strings in S_+ when parsing them in the DFA (Algorithm 4). Our goal is then to cluster the transitions as in the spirit of [151]. In their work, they perform state sequence clustering to figure out different driving behaviours, represented as timed automata. That is why given: a DFA and a positive dataset S_+, we first look which strings in S_+ use transitions, and then return the number of different *paths* taken by the strings. Note here that the accepting state of the string is part of the solution. We can then, at the termination of the evolutionary algorithm once the best individual is found, return a solution containing k sub-DFA using the k set of transitions, as explained in Algorithm 4.

Algorithm 4 Getting sub-DFA out of a DFA by transition clustering.

Input: S_+ : set of positive strings, $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$: input DFA
Output: $\mathcal{O}$: set of sub-DFA of $\mathcal{A}$
$\mathcal{O} \leftarrow \emptyset$
for $s \in S_+$ **do**
  if $\delta^*(q_0, s) \in F$ **then**
    $Q_{sub} \leftarrow \{\delta^*(q_0, p) : p \in \texttt{prefix}(s)\}$
    $\forall q \in Q,\ \forall a \in \Sigma : \delta_{sub}(q, a) \leftarrow \{\emptyset\}$ `// initialize` δ_{sub}
    $q_{curr} \leftarrow q_0$
    for $ua \in \mathit{prefix}(s)$ **do**
      $\delta_{sub}(q_{curr}, a) \leftarrow \delta^*(q_0, ua)$
      $q_{curr} \leftarrow \delta^*(q_0, ua)$
    end
    $\mathcal{O} \leftarrow \mathcal{O} \cup (\Sigma, Q_{sub}, q_0, \delta_{sub}, \delta^*(q_0, s))$
  end
end
return $\mathcal{O}$

Note here that a candidate DFA accepting none of the strings in S_+ contains no set of transitions from its initial state to an accepting state. As a consequence, the result returned by Algorithm 4 would, in such case, be the empty set. From a practical perspective, the way to use the evolutionary algorithm is to apply Algorithm 4 to the best individual returned by the EA, to gather k DFA accepting strings in S_+ and rejecting all strings in S_-.

6.7 Experiments and Results

Industrial Application Our case study has been inspired by an industrial problem related to the domain of Large Scale Printers. Recent work [141] focused on the impact of design parameters of an industrial printer on its productivity. It appeared in the study mentioned above that productivity depends on the print-job being rendered. To that end, the prior identification of the different print-job patterns is crucial in enabling printer engineers to optimize parameters related to paper flow, according to each significant type of job being printed.

We consider a dataset containing strings, each representing a print job. Each print job is composed of several pages, that can be of different media types (papers of different thickness, sizes, etc.). Thus, each string is composed of letters standing for the media-type of the printed page. We could have for instance *aaaaaabbbbbbbbbb* that would represent a print job consisting of 6 pages of type a, and 10 pages of type b.

print job	pattern	type of print job
aaaaa	a^+	homogeneous job
aaaaaaaaaa		
aaaaa . . . aaa		
abababab	$(ab)^+$	heterogeneous job
abababababababab		
abcabcabc	$(abc)^+$	
abcabcabcabcabcabc		
abcbcbcbca	$a(bc)^+a$	booklet
abcdebcdebcdea	$a^+(bcde)^+a^+$	

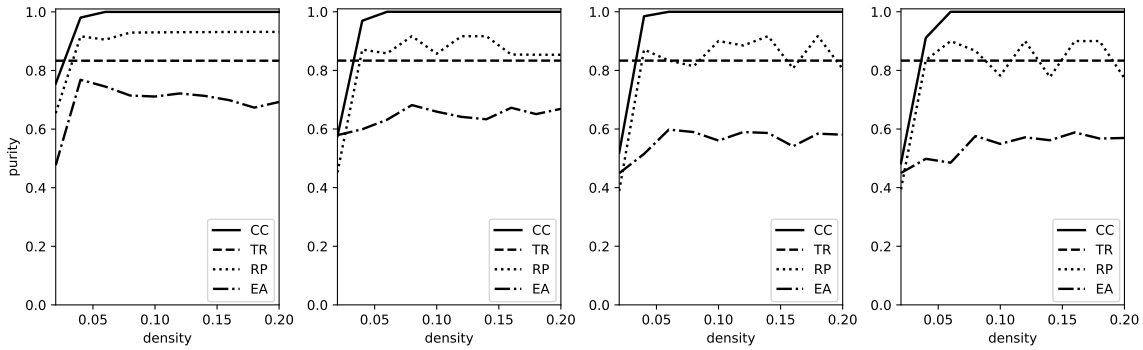
TABLE 6.2: Sample of identified print-job patterns.

Experiments We evaluated our methods on a set of 6 disjoint languages $\mathcal{I} = \{\mathcal{L}(a^+), \mathcal{L}((ab)^{\geq 2}), \mathcal{L}((abc)^+), \mathcal{L}(ab^+a), \mathcal{L}(a^+b^+), \mathcal{L}(a(bc)^+a)\}$ from which 100 strings are generated. The strings are divided into a training set, and a test set T. The proportion of strings in the training set in comparison to the strings in the test set (density) varies from 0.02 to 0.20. We consider the four methods presented in this chapter, the clustering by compression (CC), the identification of tandem repeats (TR), our modified version of RPNI (RP) and the evolutionary algorithm (EA). We denote by $\mathcal{O}$ the set of DFA returned by the methods. The results are stated in terms of purity. This represents to what extent the learned DFAs recognize a single target language. Note here that this measure is relevant in the case of disjoint languages. For each learned DFA, we count the number of strings issued from the most common of the input languages, such that

$$purity(T, \mathcal{O}) = \frac{1}{|T|} \sum_{i \in \mathcal{I}} \max_{o \in \mathcal{O}} |T \cap i \cap \mathcal{L}(o)|$$

The negative dataset S_- necessary for RP and EA is composed of $\Sigma^1 \cup \Sigma^2$ with $\Sigma = \{a, b, c\}$. CC and TR do not require any negative set. We evaluated our methods on each subset of k distinct languages for k from 2 to 5. Each experiment has been run 100 times. The number of input languages k has been given as a parameter to the learning algorithms, to return k output languages. We show in Figure 6.2a to 6.2d the purity as a function of the density, depending on the number of languages to learn k.

The method CC performs remarkably well even with small training sets, whereas TR fails to generalize repeated patterns, only having a cardinality of at least 2. For RP, we see that a dataset size of at least five strings improves the results. Concerning EA, we see that it underperforms whenever the number of DFA to learn



(A) $k = 2$ (B) $k = 3$ (C) $k = 4$ (D) $k = 5$

FIGURE 6.2: Results depending on the number of DFA to learn.

exceeds 4. Surprisingly, it seems that approaches which learn from positive examples achieve better results than those needing positive and negative examples, especially when the number of DFA to learn is too large. However, RP and TR are comparable. Our dataset and implementations are available[1].

Class of languages It is easy to see that CC, TR and RP will fail to distinguish $\{w \in \{a\}^* \mid |w| \mod 2 = 0\}$ from $\{w \in \{a\}^* \mid |w| \mod 2 = 1\}$ when they both are in S_+. Indeed, by construction of these algorithms, nothing ensures that odd or even repetitions of a pattern will be differentiated. However, they will properly recognize languages for which there is an integer n such that for all words x, y, z and integers $m \geq n$ then $xy^m z \in L$ if and only if $xy^n z \in L$, that is to say, star-free languages. However, EA offers more liberty and can possibly recognize some of the non-star-free languages (such as odd/even number of a's) since one can imagine a small dataset and a high number of clusters to find. The identification of different numbers of repetitions seems possible.

6.8 Conclusion

In this chapter, we described four methods to learn several distinct languages from text or complete presentations. From a practical perspective, we see that the gathering of more than one automaton is meaningful since a too generalized solution is not necessarily the best. We also evaluated our methods and conclude that compression is useful for grammatical inference. As further works, we would like to investigate the definition of a metric that would compare the individual DFAs provided by the different methods. Nonetheless, we think that the sole use of the predictive factor of the different automata is a good first step. Relevant heuristics or statistics to that end promises to be complicated since it is difficult to determine whether a solution is better than another because it generalizes too much/not enough, gives too many/too few sub-DFA, etc. However, depending

[1] https://gitlab.science.ru.nl/alinard/learning-multiple-dfas

on the applications, expert knowledge can decide which heuristic is relevant. We would also like to extend our techniques to deal with non-disjointness of languages. To that end, we will explore the inference of probabilistic finite automata for learning several languages. Finally, we think that we can develop active learning techniques in order to learn several systems by querying a teacher and learning more than one System Under Test (SUT). Of course, this implies to redefine Minimally Adequate Teachers (MAT), and also the queries the learner sends to the teacher.

The output of our algorithm, applied to Large Scale Printers, is further detailed in Chapter 7.

Chapter 7

An Application of Hyper-Heuristics to Flexible Manufacturing Systems

Optimizing the productivity of Flexible Manufacturing Systems requires online scheduling to ensure that the timing constraints due to complex interactions between modules are satisfied. This work focuses on optimizing a ranking metric such that the online scheduler locally (i.e., per product) chooses an option that yields the highest productivity in the long term. In this paper, we focus on the scheduling of a re-entrant Flexible Manufacturing System, more specifically a Large Scale Printer capable of printing hundreds of sheets per minute. The system requires an online scheduler that determines for each sheet when it should enter the system, be printed for the first time, and when it should return for its second print. We have applied genetic programming, a hyper-heuristic, to heuristically find good ranking metrics that can be used in an online scheduling heuristic. The results show that metrics can be tuned for different job types, to increase the productivity of such systems. Our methods achieved a significant reduction in the jobs' makespan.

7.1 Introduction

Flexible Manufacturing Systems (FMS) are composed of thousands of components, which work together to manufacture products. The timing between the different production steps can be complicated due to the wide range of products that the system can produce. Such systems require an online scheduler to ensure that the timing constraints for a single product, as well as for consecutive products, are satisfied. A Large Scale Printer (LSP, see Fig. 1.2) is such an industrial FMS that requires online scheduling. The combination of minimal and maximal time lags between different production steps, the re-entrant loop, as well as the variability of products makes this a challenging task. The design of scheduler components, as well as the design of ranking metrics, takes a significant amount of effort. In this paper, we use the LSP as a use case for automatically optimizing metrics during design time, to be used during online scheduling. The paper path of an LSP [137] consists of a Paper Input Station, an Image Transfer Station, a Re-entrant Loop with a Turn Station, and a Finish Station. A scheduler must determine an efficient order of the first and second pass printing at the Image Transfer

Station for a given sequence of sheets. This is a difficult problem to solve online due to the minimal and maximal re-entrance time, and the sequence-dependent set-up times between subsequent products [146]. The sheets of paper must not collide, and there should be sufficient time between operations on subsequent products. Leaving as little time as possible between the release of products may lead to high productivity in the short term, but if a future sheet requires large set-up times, then the Re-entrant Loop must first be emptied before that sheet can be printed. The scheduling decisions are therefore critical for optimizing the long-term productivity: for each newly entering sheet, a scheduling decision is made whether to first print a returning sheet or to release the new sheet. This decision is referred to as determining the interleaving of sheets. The second pass sheets re-enter the print section and are called re-entrant sheets. The goal is then to optimize the scheduling decisions of the print jobs, to maximize the productivity (i.e., minimize the printing time) for a document. Print jobs can be grouped into different categories, based on structural similarities. Take, for instance, multiple copies of a booklet, following a specific pattern (a cover and inside pages), then the number of pages only differs between print jobs. Finding ranking metrics that optimize for such families can yield improvements for the online scheduler.

In this work, we focus on discovering productive ranking metrics, that are used by an online scheduler to determine the most productive sequencing decision. Our contributions are: (1) an application of hyper-heuristics to Flexible Manufacturing Systems, and particularly to Large Scale Printers and (2) a modification of genetic programming approaches to ensure a fast convergence towards optimal solutions. We use genetic programming as hyper-heuristics to perform design-time optimization of the metrics that are used online to select the scheduling decisions [62]. Hyper-heuristics explore the search-space of parameters or more generally *programs*, to discover the best settings for the (scheduling) heuristic. The hyper-heuristic generates programs, representing ranking metrics, which are evaluated on a benchmark given the same scheduling framework like the one used for online scheduling.

We have experimented with two different genetic programming approaches and two benchmark aggregations. Our first approach consists of running a Genetic Program (GP) for each group in the benchmark, using user-defined initialized metrics. Our second approach collects the metrics generated for all groups generated by the first approach and uses these as the initial population of metrics for a second run. We also report the results for different aggregations; our two approaches are applied to each print job individually, and groups of structurally similar benchmarks corresponding to the print job patterns found in Chapters 5 and 6.

The remainder of this chapter is as follows: in Sections 7.2 and 7.3 we define the re-entrant flow-shop scheduling problem and discuss its relevance for the productivity of FMSs, as well as related work. In Sections 7.4 and 7.5 we present several versions of the hyper-heuristic genetic programming approach. We present then in Section 7.6 the experimental evaluation of our methods on a Large Scale Printer. Finally, we conclude and mention further work.

7.2 Problem Definition

The growing complexity of industrial systems leads to new issues related to scheduling and control. Solving these combinatorial problems can be difficult, and online solutions are generally based on heuristics, representing a good trade-off between computation time and solution quality. These heuristics are, in practice, built upon expert knowledge. However, the acquisition of such human expertise is costly (over the years) and may be biased towards particular solutions.

Therefore, there has been an increase in attention for automatically generating new heuristics for search problems [33, 62]. Approaches that heuristically find new heuristics are called hyper-heuristics [34]. Hyper-heuristic approaches, such as tabu search [31], genetic programming [33], and simulated annealing [11], have been successfully applied to finding better heuristics for bin-packing [125], vehicle routing problems [144], scheduling (constructively or with improvement steps) [32], as well as improving strategies for reinforcement learning [112].

7.2.1 Use Case

The use case of this paper, an LSP, is modelled as a re-entrant flow-shop scheduling problem. We repeat a shortened version of the problem definition in this section. For a comprehensive discussion of the problems components, we refer the reader to [117].

Definition 7.2.1 (re-entrant flow shop)**.** *A re-entrant flow shop with sequence-dependent set-up times and relative due dates is a tuple* $(M, J, r, O, \phi, P, S, SS, D)$*. M is a set of machines that executes operations. J is a sequence of jobs. Every job j in J is a sequence of r operations $\langle o_{j,1}, \ldots, o_{j,r} \rangle$ that need to be executed. The operations O of the flow shop are the union of the operations of its jobs: $O = \{o_{j,k} | j \in J, k \in \{1, \ldots, r\}\}$. We write $o_{j,k}$ to denote the k-th operation of the j-th job. ϕ is the re-entrance vector $\langle \mu_1, \ldots, \mu_r \rangle$ with $\mu_i \in M$ of r machines. $\phi(i)$ denotes which machine executes the i-th operation of each job is mapped. The processing times of operations are $P : O \to \mathbb{R}_{>0}$. Set-up times that occur regardless of the sequence of operations on a machine are a partial function $S : O \times O \mapsto \mathbb{R}_{\geq 0}$, where $S(o_x, o_y)$ is the minimal time needed between completion of o_x and beginning of o_y. Sequence-dependent set-up times are $SS : O \times O \to \mathbb{R}_{\geq 0}$, where $SS(o_x, o_y)$ is the minimal time needed from completion of o_x to beginning of o_y. Relative due dates are a partial function $D : O \times O \mapsto \mathbb{R}_{>0}$.*

Definition 7.2.2. *A schedule $B : O \to \mathbb{R}_{\geq 0}$ for a flow shop describes begin times for all operations; $C(o) = B(o) + P(o)$ denotes the time that operations complete. The following constraints need to be satisfied: Machine $\mu \in M$ executes at most one operation at a time. The sequence-independent set-up times of S between each pair of operations o_x and o_y in the domain of S must be met: $B(o_y) \geq C(o_x) + S(o_x, o_y)$. If no other operation begins between o_x and o_y on the same machine, then the sequence-dependent set-up time between them must hold: $B(o_y) \geq C(o_x) + SS(o_x, o_y)$. If a due date $D(o_x, o_y)$ from o_x to o_y is defined, then it imposes a due date on o_y s.t. $B(o_y) \leq B(o_x) + D(o_x, o_y)$. A schedule is feasible only when all of these constraints are satisfied.*

The goal of the optimization problem is to minimize the makespan of a schedule B, which is the latest completion time of any operation: $makespan(B) = \max_{o \in O} B(o) + P(o)$

The scheduling freedom consists of choosing the ordering and timing of the products at the re-entrant machine (the Image Transfer Station) that satisfies the sequence-dependent set-up times and does not violate the relative due dates (i.e. the maximum travelling time between two prints). An LSP can be modelled as a 3-machine flow shop with four operations per job (i.e., a sheet), re-entrance vector $\langle \mu_1, \mu_2, \mu_2, \mu_3 \rangle$, and one job for each sheet that needs to be printed [146].

The heuristic makes its decision on a per-sheet basis. For each subsequent sheet, its re-entrant (second pass) operation is inserted at several positions of a previously determined ordering of first and second passes. To determine whether a given sequencing option is (locally) feasible, a longest-path algorithm is used to determine the earliest possible realization times of the operations of a series of products, taking into account all active timing constraints. The scheduling heuristic then uses a ranking metric to select the scheduling option that lead to the best long-term productivity. The BHCS heuristic of [117] is currently the best on-line dispatching rule heuristic. This paper aims to improve the ranking metric used in the dispatching rule framework of BHCS.

Typical ranking metrics for this problem include information such as the realization time for executing the second pass of the currently scheduled sheet, the number of operations that remain to be scheduled, the delay of an event due to a scheduling decision (adding the current sheet's second pass into the previously determined sequence).

7.2.2 Ranking metric components and structure

The re-entrant operations are sequenced one at a time. Such an operation is denoted by $O_{j,p}$, where j is the job id, and $p \in \{1, 2\}$ is the pass id of the operation. The operation immediately following $O_{x,y}$ is denoted by $O_{w,z} = NEXT(O_{x,y})$, which is influenced by the schedulers' decisions. The earliest realization time of $O_{x,y}$ (for a given sequence) is denoted as $t_{x,y}$. The earliest realization time without the scheduling option under consideration is denoted by $t'_{x,y}$. A limit job L is found for each scheduling option; any option for which the limit job's first pass precedes the current job's second pass is a priori infeasible [117].

Note that X0, X1, X3, X4, X6, and X8 have the same value for all options in a single scheduling iteration. These values do change per scheduling iteration, and may therefore still contain information regarding the distinction of scheduling options.

The ranking metric used in [117] has been determined empirically. It uses normalization of these components over the set of scheduling options P. The ranking

TABLE 7.1: Ranking metric's variables. The currently scheduled second-pass operation is $O_{j,p}$.

Variable name	Component
X0	# sheets in the print job
X1	# operations per sheet
X2	L
X3	j
X4	p
X5	$t_{j,p}$
X6	$t'_{j,p}$
X7	# operations between $O_{j,p-1}$ and $O_{j,p}$
X8	Maximum of X7 over all options of the current iteration
X9	x of $O_{x,y} = NEXT(O_{j,p})$
X10	y of $O_{x,y} = NEXT(O_{j,p})$
X11	$t_{x,y}$ for $O_{x,y} = NEXT(O_{j,p})$
X12	$t'_{x,y}$ for $O_{x,y} = NEXT(O_{j,p})$

metric for an option $p \in P$ is a mix of constants, min, max, multiplications, additions, subtractions and divisions:

$$\begin{aligned} rank(p) = 0.3 & \frac{X5_p - min_{p' \in P}(X5_{p'})}{max_{p' \in P}(X5_{p'}) - min_{p' \in P}(X5_{p'})} \\ + 0.6 & \frac{X11_p - min_{p' \in P}(X11_{p'})}{max_{p' \in P}(X11_{p'}) - min_{p' \in P}(X11_{p'})} \\ + 0.1 & \frac{X7_p - min_{p' \in P}(X7_{p'})}{max_{p' \in P}(X7_{p'}) - min_{p' \in P}(X7_{p'})} \end{aligned}$$

Besides the calculated parameters, the following functions are allowed: minimum, maximum, addition, subtraction, multiplication, `asapst`, square root, absolute value, logarithm and negation. The limitation to these functions is domain dependent. The `asapst` function is a domain specific function that takes a job j and a pass id p, and returns the earliest realization time of that operation, i.e. $t_{j,p}$.

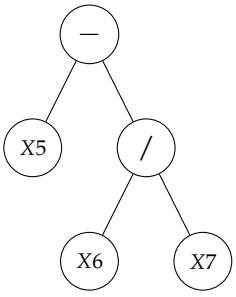


FIGURE 7.1: Ranking metric as expression tree.

Note here that the structure of the ranking metric can be seen as an expression tree. Nodes correspond to the functions and leaves to either variables or constants. This is this structure that will, later, be manipulated by the genetic program. An example of such a tree-like structure is shown in Fig. 7.1.

7.3 Genetic Programming

The approach we use is characterized as an off-line heuristic learning approach for a constructive heuristic [34]. To that end, we focus on genetic programs that act as hyper-heuristics for generating heuristics. GPs are suitable for the generation of heuristics (we refer in this regard to the survey paper of [33]).

Evolutionary Algorithms [10] are Artificial Intelligence methods based on heuristics which are particularly suitable to automatically learn *models* (e.g., trees [29], graphs [52], equations [110], etc ...). They are used in a broad variety of domains e.g. learn the optimal shape of tree-like NASA's antennas [71], robot designs [39] and even the internal structure of artificial neural networks [84]. While most evolutionary algorithms focus on solving single-objective problems, there are extensions to solve multi-objective problems [47]. Genetic Programs [134] are a specific type of Evolutionary Algorithms, where programs/heuristics compute solutions, while Evolutionary Algorithms in a broad sense directly evolve solutions to a problem.

Many different techniques have been developed in the past few years, among others, the combination of scheduling rules instead of considering a set of rules separately, in order to enlarge the search space [54, 46]; genetic algorithms to search a space of sequences of heuristic choices [53, 124]; genetic algorithm to solve real-world scheduling and delivery problems [67]; evolutionary algorithm to learn successful heuristics from previous ones [49, 50]. The latter caught our attention: previously learned heuristics (automatically or manually, using domain expertise) could be applied to instances from a given problem after a learning phase. In the context of GPs, this existing knowledge can, therefore, be used as an initial population of metrics.

7.4 Learning Approach

One of the critical concepts of GP is to formalize what makes a solution better than another, i.e., writing a *fitness function* which takes any proposed solution and returns a *score* for that solution. In our case, we want a ranking metric to lead to the highest productivity as possible, that is to say, to minimize the makespan of the corresponding print job.

Then, the GP iteratively optimizes a *population* of candidate solutions, by randomly mutating and combining (i.e., applying *genetic operators* to) solutions from a population, such that the fitness of the *best solution* per population improves in time:

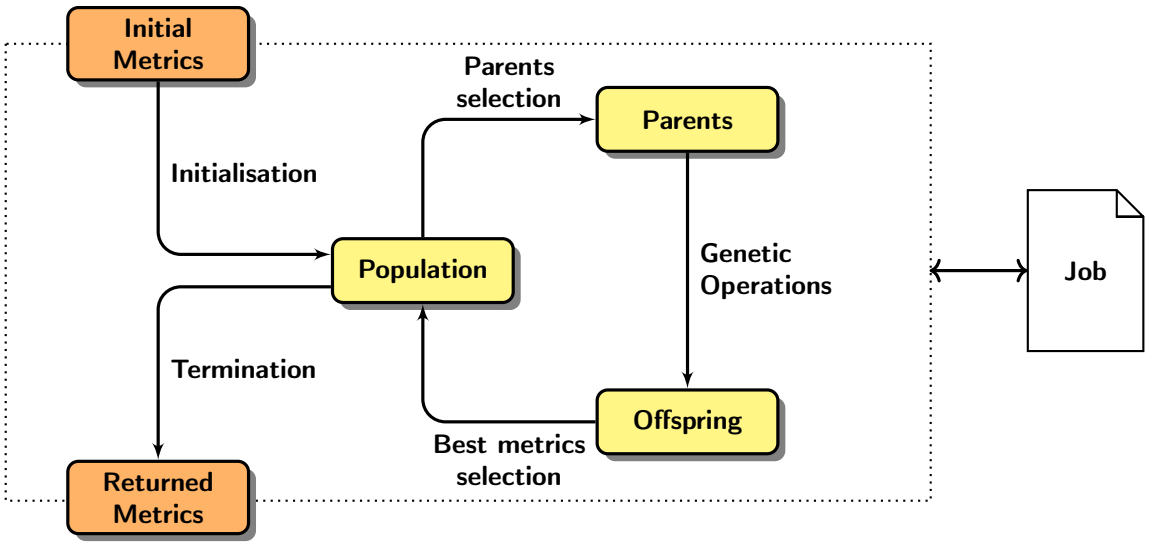


FIGURE 7.2: Genetic Program optimizing an individual print job (GP-i).

1. *Initialization*: the initial population contains a set of possible ranking metrics;

2. *Selection*: the selection of the best ranking metrics in the population is based on their fitness values.

3. *Mutation and recombination*: genetic operations are performed on the selected ranking metrics with a fixed probability, and generate new ranking metrics.

4. *Evaluation*: the fitness of each new schedule is evaluated.

5. *Replacement*: high-fitness ranking metrics from the new generation replace low-fitness ranking metrics from the previous generation.

As shown in Fig. 7.2, the last four steps are repeated until a maximum number of allowed iterations is reached: then, the best solution in the population is returned.

Many variations or parameters can be tuned in GP, such as the definition of a good fitness measure to evaluate candidate programs; the termination criteria, standing for how a solution needs to be good enough to consider returning it; the terminal set, meaning what symbols are allowed in the leaves of the tree-like structure of the program; the function set, defining the functions at the non-terminal nodes of the program; the initial population, which can be user-defined or randomly generated; other GP execution parameters such as (but not limited to) selection method, population size, mutation methods, probability of occurrence of genetic operators, allowing elitist strategies, maximum allowed size/depth of the programs, etc.

7.4.1 The Initialization of the Genetic Program

We start with an initial population consisting of commonly used ranking metrics.

- $X5 - \frac{X6}{X7}$
- $|X5|$
- $\text{asapst}(X2, X1)$
- $X5 + X6$
- $X5 - X6$
- $X11 - X12$
- $\frac{X7}{X8}$
- $(X5 - X6 + X11 - X12) - \frac{X7}{X8}$

7.4.2 Selection

The selection of individuals that will undergo genetic operations is an important task: if the selection method overly selects fit individuals, the population may converge too quickly to a uniform solution, while not exploring the search space widely enough. On the contrary, a selection method not selecting enough fit individuals, will not correctly exploit the information present in the fittest individuals.

We used here a *tournament* selection. From the population, a smaller subset is selected at random to compete. The fittest individual in this subset is then selected to move on to the next generation. Depending on the size of the tournament subset, fitter programs can tend to be found quickly, and the evolution process tends to converge to a solution in less time (large tournament size) or maintain more diversity in the population and find a better solution but for a longer runtime (small tournament size).

7.4.3 Terminal and Function Sets

The terminal set is composed of all variables names as described in Table 7.1, plus constants ranging over $\mathbb{R}$. The function set is composed of 5 chosen operations, which are addition, multiplication, subtraction, division and `asapst`.

7.4.4 Genetic Operators

We define stochastic genetic operators as follows:

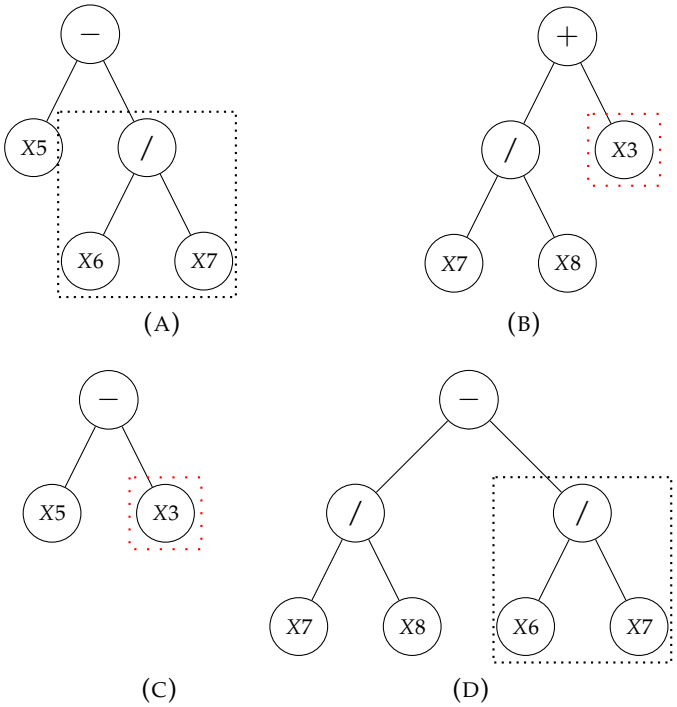


FIGURE 7.3: Crossover between individuals (A) and (B) and resulting children (C) and (D).

1. *Crossover*: Crossover takes a metric M_1 and selects a random subtree from it to be replaced. A second metric M_2 also has a subtree selected at random, and this is inserted into the original parent to form an offspring in the next generation and vice versa. An example of this genetic operator is shown in Fig. 7.3.

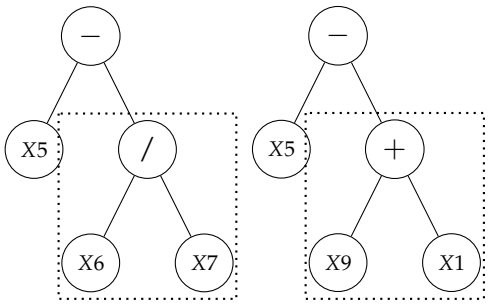


FIGURE 7.4: Subtree mutation.

2. *Subtree Mutation*: Subtree mutation takes a metric M and selects a random subtree from it to be replaced. A donor subtree is generated at random, and this is inserted into the original parent to form an offspring in the next generation. An example of this genetic operator is shown in Fig. 7.4.

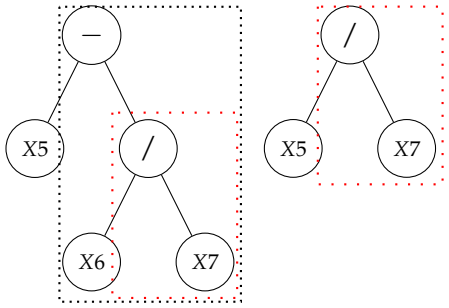


FIGURE 7.5: Hoist mutation.

3. *Hoist Mutation*: Hoist mutation takes a metric M and selects a random subtree from it. A random subtree of that subtree is then selected, and this is *hoisted* into the original subtrees location to form an offspring in the next generation. An example of this genetic operator is shown in Fig. 7.5.

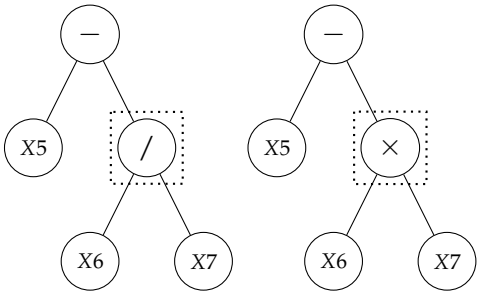


FIGURE 7.6: Point mutation.

4. *Point Mutation*: Point mutation takes a metric M and selects random nodes from it to be replaced. Other terminals and functions replace terminals that require the same number of arguments as the original node. The resulting tree forms an offspring in the next generation. An example of this genetic operator is shown in Fig. 7.6.

7.4.5 Fitness Function

We use the makespan of a job J for a ranking metric r as its *fitness*. In other words,

$$fitness(r) = makespan_r(J) = \max_{o_{j,p} \in O} \left(t_{j,p}\right)$$

7.5 Further Approaches

In this section, we describe three extensions of our genetic program. The first one consist of first running the GP on the initial population described in Section 7.4.1, then reusing the resulting metrics as the initial population for a second run of the GP. The second and third ones consist of evolving metrics such that the fitness is not a function of a given job, but a set of jobs to schedule.

7.5.1 Reseeding the Genetic Program

The Genetic Program, as depicted by Fig. 7.2 tries to optimize a schedule for a given job. The extension we propose is to first run this optimization process once, for the given job. We gather at the end of this process, a set of metrics for this job. The next step of the process is the following: a second GP is run, still in order to optimize this same job, with input population all the metrics of the job to optimize gathered during the first step.

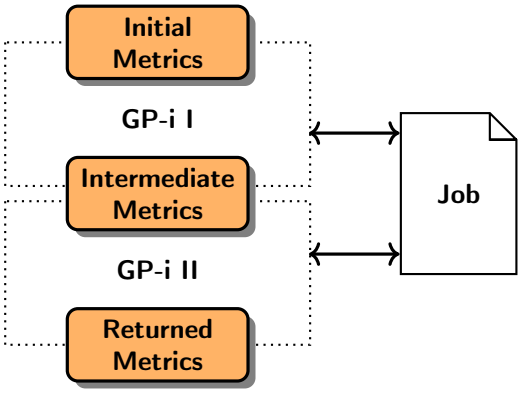


FIGURE 7.7: Genetic Program optimizing an individual print job, using "re-seeding" (GP-ir).

This operation of "Re-seed" allows us to reuse previously found good metrics for the job, so that good metrics will tend to be selected, bred and returned at the end of this second process. The whole process of GP-ir is described in Fig. 7.7.

7.5.2 Global Solutions

While the previous methods consider a fitness function of the GP as a function of 1 job to optimize, we propose two methods trying to optimize a set of jobs at once.

The first one, Genetic Programming to solve a group G of print jobs (GP-g), starts with the initial population as described in Section 7.4.1. Then, unlike GP-i, the fitness function is redefined as the sum of all the makespans of the set of jobs to schedule, such that

$$fitness(r, G) = \Sigma_{J \in G} makespan_r(J)$$

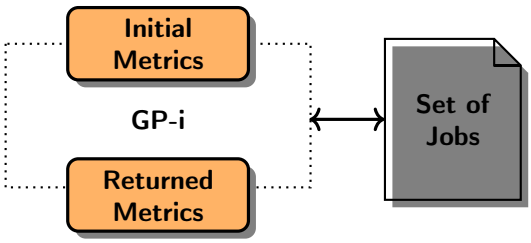


FIGURE 7.8: Genetic Program optimizing a group of print jobs (GP-g).

The task becomes then to optimize a single metric common to a set of jobs. This is a particularly interesting approach in the context of similar jobs, i.e. sharing similar operations or patterns of operations, such as the ones found in Chapters 5 and 6. This process is described in Fig. 7.8.

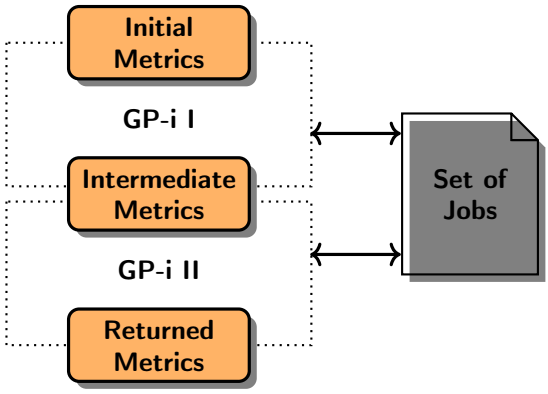


FIGURE 7.9: Genetic Program optimizing a group of print jobs, using "re-seeding" (GP-gr).

The second one, Genetic Program to solve a group of print jobs using "re-seeding" (GP-gr), is a mixture of GP-ir and GP-g. It tries to optimize a ranking metric common to a set of jobs, going through 2 steps: GP-g then GP-g re-seeded with the results of the first iteration. This process is described in Fig. 7.9.

7.6 Experimental Evaluation

In this section, we describe the experimental set-up and compare the makespan of the generated schedules through our different methods. We consider in this paper a benchmark for Large Scale Printers (such as the one in Fig. 1.2). We have implemented our 4 genetic programs, GP-i, GP-ir, GP-g and GP-gr, using the python *gplearn* API[1]. BHCS is implemented as a single-thread program. We reused the benchmark used in [117] consisting of 85552 sheets in 684 print requests. Each print request is taken from one of the following categories:

[1]https://gplearn.readthedocs.io/en/stable/index.html

H Homogeneous: repetition of one sheet type

RA Repeating A: repetition of one sheet type followed by another sheet type

RB Repeating B: repetition of one to three sheets of a type followed by another sheet type

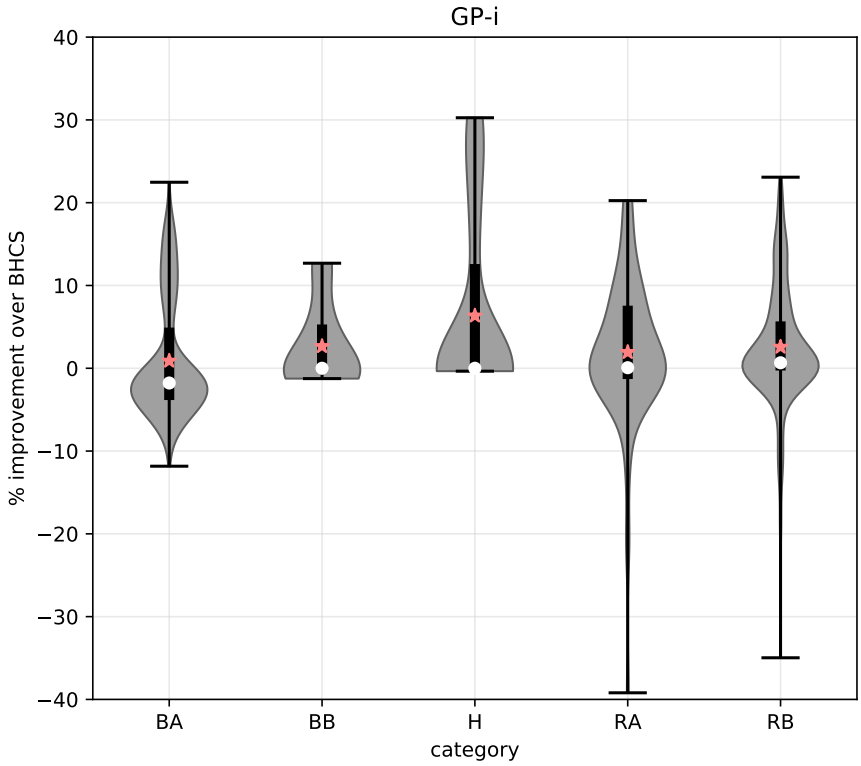
BA Block A: 5 blocks of 5 different sheet types, each block contains 10 or 20 sheets of the same type

BB Block B: 5 blocks with two alternating sheet types, each block contains 5 to 25 sheets of the same type

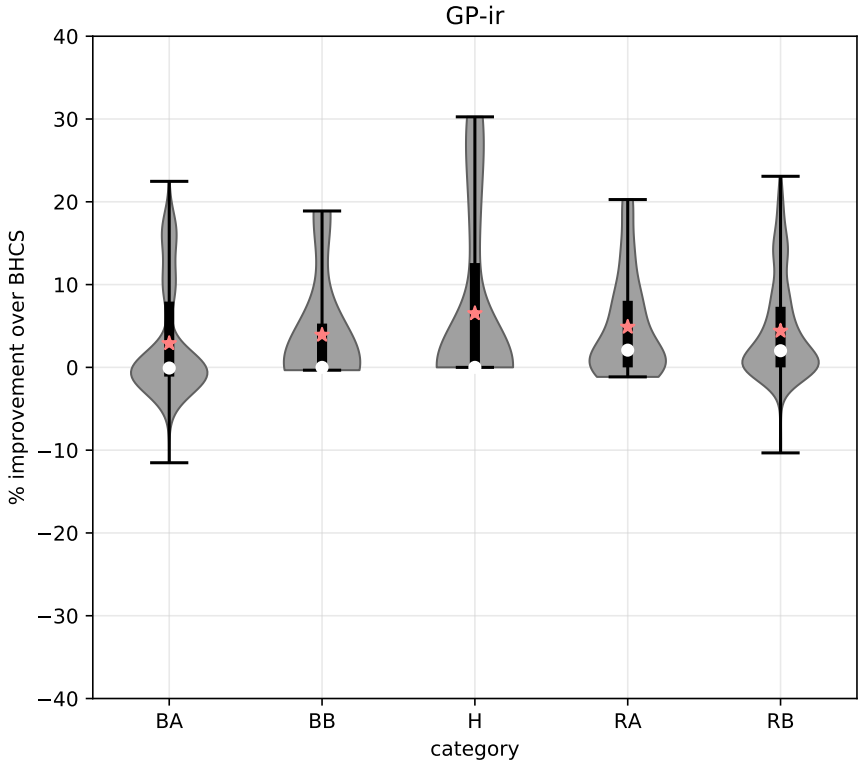
The GPs were parametrized as follows:

- maximum number of iterations: 100
- population size: 100
- tournament size: 10
- crossover probability: 0.7
- subtree mutation probability: 0.1
- hoist mutation probability: 0.1
- point mutation probability: 0.1

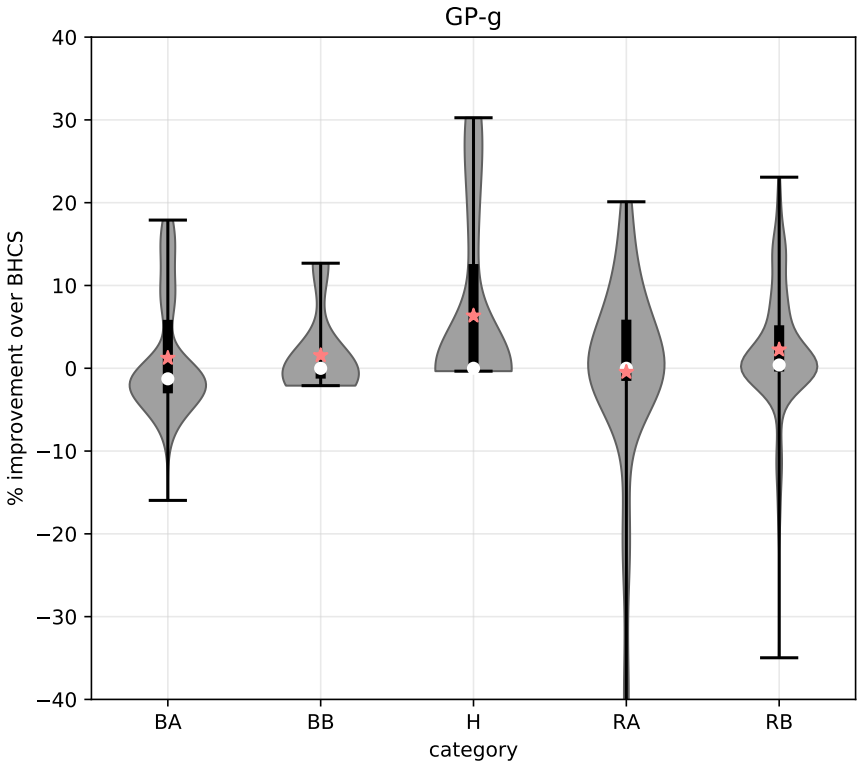
Note here that the population size is voluntarily kept low: indeed, it has been shown that too big populations are not always helpful in order for the GP to converge faster towards an optimal solution [38]. Large population sizes also have deleterious effects on the time efficiency of the GP: indeed, in case of a too high calculation complexity of the fitness function, the run time may dramatically blow up. In order to speed up the process, results for given metrics were cached, so the recomputation of previously seen individuals is not required. For the methods GP-g and GP-gr, up to 100 processors were used in parallel for the computation of ranking metrics for a set of jobs.



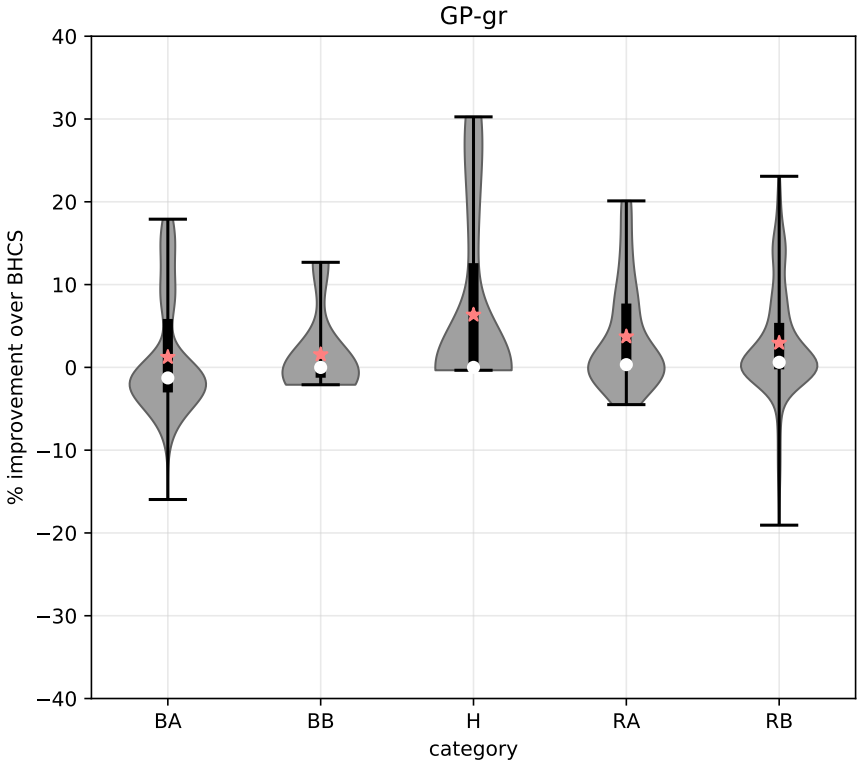
(A) Results for GP-i.



(B) Results for GP-ir.



(C) Results for GP-g.



(D) Results for GP-gr.

FIGURE 7.9: Makespan improvements of our methods compared to BHCS (in %).

We compare the makespans obtained with the makespans of schedules generated by BHCS in Fig. 7.9. For each job category, we show the median improvement (white dot), average improvement (red star), as well as the interquartile range (black box), and the probability density of the data at different values. We also calculated the runtime of the different algorithms, per benchmark, in Fig. 7.10.

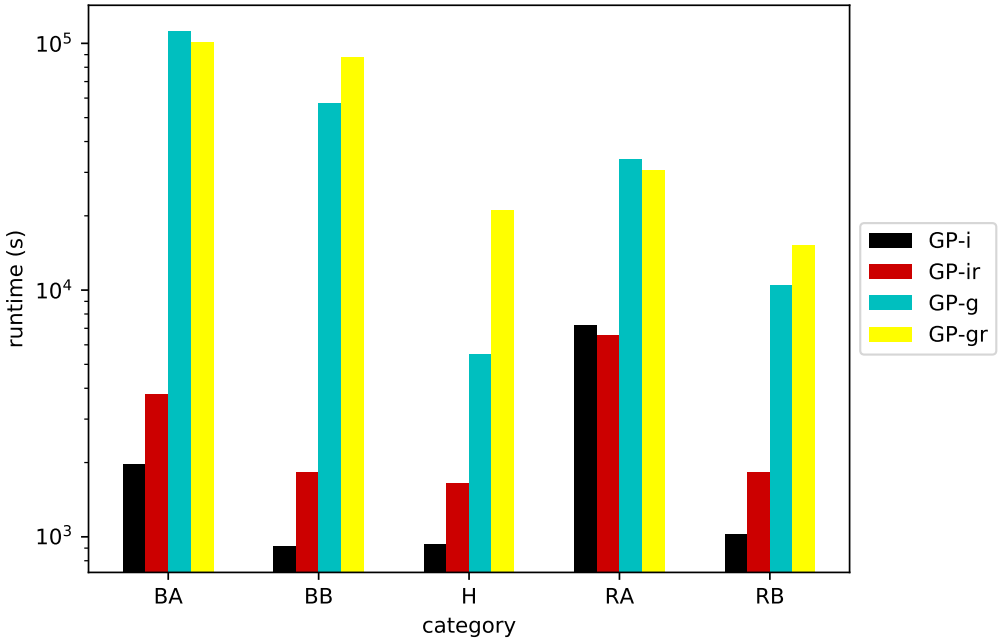


FIGURE 7.10: Runtime of the different algorithms (in seconds).

We can see here that for all methods and job types (except GP-g for RA), the average obtained makespan compared to BHCS is improved. In some cases (jobs H), the found heuristics improve the makespan for up to 30.26%. Concerning the median improvement close to 0%, it shows that our methods improve the printing time of the jobs in half of the cases in our benchmark. These are encouraging results: whenever a job is not improved, we can instead fall back to the metric provided by BHCS. So, improving the productivity of half of the jobs is a good result.

7.7 Conclusion and Future Work

We presented a new method for the improvement of metrics used to make scheduling decisions for product sequences in a Flexible Manufacturing System. To that end, we employed hyper-heuristics to find these productive metrics. To the best of our knowledge, our work is the first contribution using hyper-heuristics for improving the productivity of FMSs and LSPs. In the case study we considered, our re-seeded genetic program achieved an average improvement of 3.10% in terms of schedule completion time. Moreover, out of 684 cases, we could improve 396

of the print job schedules. For some cases, the execution time of the print jobs has been reduced by up to 13.5 seconds (30.3%), which is a significant productivity improvement in an industrial context.

Our re-seeding of the initial population based on the schedules of a set of jobs shows that some jobs share common productive metrics. However, the running time of a Genetic Program prevents its online use in practice. Nonetheless, the schedules obtained can be, once obtained, injected in the online scheduling of jobs as a knowledge base. The integration of such metrics will lead to an improvement in the productivity of FMSs. Indeed, whenever a print job has to be scheduled, the schedules of similar print jobs in the database in terms of job patterns [91, 93, 95, 145] can be reused. In further work, we will focus on the integration of good ranking metrics for similar print job patterns.

Chapter 8

Conclusion

This thesis spans several areas of research, among others machine learning, grammatical inference, model checking and reliability engineering. They all comprise applications to Cyber-Physical Systems, which are thoroughly described in the manuscript. The contributions of this thesis aim at improving the efficiency, the reliability and the safety of CPS. To that end, we use a set of different model learning techniques. An exciting feature of all these different contributions is the variety of models they consider: Fault Trees, Automata, Tree-like Ranking Metrics, and more "classical" Machine-Learning classifiers. In all cases, we broke away from the popular neural networks, which suffer from a certain lack of expressiveness we needed in all our applications. We present hereinafter a summary of the contributions and results, as well as directions for future work.

Learning Maintenance Policies for CPS

In Chapter 2, we described a new method for dynamic maintenance scheduling. The major novelty of our method is to put together machine learning and model checking by:

1. using the decision of a classifier in order to put forward or postpone when maintenance should be triggered, i.e., how to update the scheduling of maintenance for a CPS.

2. using a model checking tool to simulate the benefits of such a method.

We claim that maintenance actions are frequently scheduled with intervals that are suboptimal. We also note that the price to pay in order to do less maintenance is a slight increase of the expected failure rate.
Concerning further work, looking into the use of fault trees [126] might be interesting, to combine at the same time existing expert knowledge expressed as reliability models and model checking. We can also enhance the scheduler by taking into account the timing occurrences of anomalies [97]. Finally, we could orientate the choice of the machine learning techniques used towards stream mining tools and algorithms [18], which would additionally offer the possibility of updating the failure behaviour model the more new labelled data are available. Then, it could be possible to deal with unseen events or combination of parameters.

Learning of Fault Trees for CPS

In Chapters 3 and 4, we present methods for automated generation of fault trees from Boolean observational data. In the domain of reliability engineering, our contributions are then two-fold:

1. an evolutionary algorithm for accurate inference of fault trees. We defined a set of genetic operators specific to the formalism of fault trees.

2. a framework for the Bayesian learning of fault trees. We defined a set of translation rules from previously learnt Bayesian Networks from data, to the desired fault trees.

In both cases, we could experimentally compare the accuracy of our methods to classical machine learning techniques, such as classifiers. Our methods also showed to be particularly robust to noisy data, even though improvement still has to be made.
About future research, the obvious extension of this work is toward dynamic fault trees [51]. This powerful formalism extends one of the static fault trees to allow time dependencies, with the addition of extra gates, such as:

- *functional dependency* gate: models how a trigger event can cause dependent components failures.

- *priority AND* gate: models how a specific succession of components failures leads to a higher level failure.

- *spare* gate: models the failure of components comprising primary and alternate (spare) components (e.g. 4+1 wheels of a car, spare batteries, etc...)

An identified track to explore is the prior learning of a dynamic Bayesian Network from data, and the translation of such a dynamic Bayesian Network to a dynamic fault trees with appropriate rules. Dynamic Bayesian Networks are known to be efficiently learnable from data [104], hence being an appropriate step towards the learning of dynamic fault trees.

Learning Unions of Models for CPS

In Chapters 5 and 6, we identified a new task in grammatical inference, as the learning of unions of languages. Indeed, it appeared that in our case studies, a too generalized solution is not necessarily the best. We also identified two cases: on the one hand, the learning of unions of k-testable languages and, on the other hand, the learning of unions of regular languages. We list here our contributions. Specifically concerning k-testable languages:

1. we defined a Galois connection characterizing k-testable languages.

2. we described an efficient algorithm to learn unions of k-testable languages that results from this Galois connection.

3. we provided guarantees concerning the learnability of unions of k-testable languages.

Concerning regular languages:

4. we described four methods to learn several distinct languages from text or complete presentations. They comprise clustering by compression, modification of state-merging techniques, an evolutionary algorithm and identification of tandem patterns.

5. we conclude that compression is useful for grammatical inference.

A direct application of learning unions of languages has been thoroughly described in Chapters 5, 6 and 7, where the identification of print job patterns is crucial in order to improve the productivity of certain print jobs sent to a Large Scale Printer.
As further work, we would like to provide learning guarantees concerning regular languages. It has been shown, however, that unions of regular languages are not learnable in the limit. In this direction, only trying to find under which assumptions (knowing the number of languages to learn, the size of the automaton(a), assumptions on disjointness, etc.) learnability can be guaranteed seems suitable. We will explore the inference of probabilistic finite automata for learning several languages. Finally, we think that we can develop active learning techniques in order to learn several systems by querying a teacher and learning more than one target languages. Actively learning unions of languages would have interesting applications, for instance, in the learning of network protocols. Consider a setting where several machines implement different versions of a network protocol, but it is not clear which version is implemented on what machine. Then, the learning algorithm would have to make an on-the-fly distinction of what version clusters of machines are implementing and to learn the related models. Putting in place a technique of that nature would have several implications, such as to redefine Minimally Adequate Teachers (MAT), and also the queries the learner sends to the teacher.

Learning of Heuristic Schedulers for CPS

Finally, in Chapter 7, we presented a new method for the improvement of metrics used to make scheduling decisions for product sequences in a special case of CPS, Flexible Manufacturing Systems. To that end, we employed hyper-heuristics (in particular, genetic programming) to find these productive metrics. The contribution to the domain of scheduling controllers is two-fold:

1. the application of hyper-heuristics to Large Scale Printers. In particular, we could show that some jobs share common productive metrics.

2. several modifications of genetic programming approaches to ensure a fast convergence towards optimal solutions. These modifications include: 1.the collection of metrics generated by a first run of the genetic program and using

these as the initial population of metrics for a second run. 2. applying the hyper-heuristic to groups of structurally similar benchmarks.

In our experiments, we could show that for families of similar jobs, the same ranking metrics can be reused to improve the productivity of Large Scale Printers. However, the running time of a Genetic Program avoids its online use in practice. The schedules obtained can be, once obtained, injected in the online scheduling of jobs as a knowledge base. The integration of such metrics will lead to an improvement in the productivity of FMSs for some cases, which is left for future work.

Bibliography

[1] Yasmina Abdeddaïm, Eugene Asarin, and Oded Maler. "Scheduling with timed automata". In: *Theoretical Computer Science* 354.2 (2006), pp. 272–300.

[2] National Highway Traffic Safety Administration. *U.S. Department of Transportation Releases Policy on Automated Vehicle Development*. [Online; accessed 22-May-2019]. 2013. URL: https://www.transportation.gov/briefing-room/us-department-transportation-releases-policy-automated-vehicle-development.

[3] Alfred V Aho and Jeffrey D Ullman. *The theory of parsing, translation, and compiling*. Vol. 1. Prentice-Hall Englewood Cliffs, NJ, 1972.

[4] David J Allen. "Digraphs and fault trees". In: *Industrial & engineering chemistry fundamentals* 23.2 (1984), pp. 175–180.

[5] Rajeev Alur and David L Dill. "A theory of timed automata". In: *Theoretical computer science* 126.2 (1994), pp. 183–235.

[6] Dana Angluin. "Learning regular sets from queries and counterexamples". In: *Information and computation* 75.2 (1987), pp. 87–106.

[7] Alberto Apostolico and Franco P. Preparata. "Optimal off-line detection of repetitions in a string". In: *Theoretical Computer Science* 22.3 (1983), pp. 297–315.

[8] Ali Arab, Napsiah Ismail, and Lai Soon Lee. "Maintenance scheduling incorporating dynamics of production system and real-time information from workstations". In: *Journal of Intelligent Manufacturing* 24.4 (2013), pp. 695–705.

[9] Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The internet of things: A survey". In: *Computer networks* 54.15 (2010), pp. 2787–2805.

[10] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[11] Ruibin Bai and Graham Kendall. "An investigation of automated planograms using a simulated annealing based hyper-heuristic". In: *Metaheuristics: Progress as real problem solvers*. Springer, 2005, pp. 87–108.

[12] Mike Baker and Dominic Gates. "Lack of redundancies on Boeing 737 MAX system baffles some involved in developing the jet". In: *The Seattle Times* (2019). [Online; accessed 6-June-2019]. URL: https://www.seattletimes.com/business/boeing-aerospace/a-lack-of-redundancies-on-737-max-system-has-baffled-even-those-who-worked-on-the-jet/.

[13] Johan Bengtsson and Wang Yi. "Lectures on Concurrency and Petri Nets: Advances in Petri Nets". In: ed. by Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg. Springer Berlin Heidelberg, 2004. Chap. Timed Automata: Semantics, Algorithms and Tools, pp. 87–124.

[14] Johan Bengtsson et al. "Hybrid Systems III: Verification and Control". In: ed. by Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag. Springer Berlin Heidelberg, 1996. Chap. UPPAAL — a tool suite for automatic verification of real-time systems, pp. 232–243.

[15] Jean-Paul Benzécri. "Construction d'une classification ascendante hiérarchique par la recherche en chaîne des voisins réciproques". In: *Les cahiers de l'analyse des données* 7.2 (1982), pp. 209–218.

[16] Joseph Berkson. "Application of the logistic function to bio-assay". In: *Journal of the American Statistical Association* 39.227 (1944), pp. 357–365.

[17] Geert Jan Bex et al. "Inference of concise DTDs from XML data". In: *Proceedings of the 32nd international conference on Very large data bases*, pp. 115–126.

[18] Albert Bifet et al. "MOA: Massive Online Analysis". In: *J. Mach. Learn. Res.* 11 (2010), pp. 1601–1604.

[19] MW Birch. "The detection of partial association, I: the 2×2 case". In: *Journal of the Royal Statistical Society. Series B (Methodological)* (1964), pp. 313–324.

[20] Benjamin Bittner et al. "The xSAP safety analysis platform". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2016, pp. 533–539.

[21] Andrea Bobbio et al. "Comparing fault trees and bayesian networks for dependability analysis". In: *International Conference on Computer Safety, Reliability, and Security*. 1999, pp. 310–322.

[22] Andrea Bobbio et al. "Improving the analysis of dependable systems by mapping fault trees into Bayesian networks". In: *Reliability Engineering & System Safety* 71.3 (2001), pp. 249–260.

[23] P Bonissone et al. "Equivalence and synthesis of causal models". In: *Uncertainty Artificial Intelligence* (1991), pp. 255–270.

[24] Taylor L Booth. *Sequential machines and automata theory*. Wiley, 1967.

[25] Hichem Boudali and Joanne Bechta Dugan. "A discrete-time Bayesian network reliability modeling and analysis framework". In: *Reliability Engineering & System Safety* 87.3 (2005), pp. 337–349.

[26] Patricia Bouyer, Thomas Brihaye, and Nicolas Markey. "Improved undecidability results on weighted timed automata". In: *Information Processing Letters* 98.5 (2006), pp. 188–194.

[27] Marco Bozzano and Adolfo Villafiorita. "The FSAP/NuSMV-SA safety analysis platform". In: *International Journal on Software Tools for Technology Transfer* 9.1 (2007), p. 5.

[28] Marco Bozzano et al. "COMPASS 3.0". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tomáš Vojnar and Lijun Zhang. Cham: Springer International Publishing, 2019, pp. 379–385.

[29] Doina Bucur et al. "The impact of topology on energy consumption for collection tree protocols: An experimental assessment through evolutionary computation". In: *Applied Soft Computing* 16 (2014), pp. 210 –222.

[30] Peter Bulychev et al. "UPPAAL-SMC: Statistical model checking for priced timed automata". In: *arXiv preprint arXiv:1207.1272* (2012).

[31] Edmund K Burke, Graham Kendall, and Eric Soubeiga. "A tabu-search hyperheuristic for timetabling and rostering". In: *Journal of heuristics* 9.6 (2003), pp. 451–470.

[32] Edmund K Burke et al. "A graph-based hyper-heuristic for educational timetabling problems". In: *European Journal of Operational Research* 176.1 (2007), pp. 177–192.

[33] Edmund K Burke et al. "Exploring hyper-heuristic methodologies with genetic programming". In: *Computational intelligence*. Springer, 2009, pp. 177–201.

[34] Edmund K Burke et al. "Hyper-heuristics: A survey of the state of the art". In: *Journal of the Operational Research Society* 64.12 (2013), pp. 1695–1724.

[35] Alan Burns. "How to Verify a Safe Real-Time System: The Application of Model Checking and Timed Automata to the Production Cell Case Study". In: *Real-time systems* 24.2 (2003), pp. 135–151.

[36] Karen L Butler. "An expert system based framework for an incipient failure detection and predictive maintenance system". In: *Proceeding of the International Conference on Intelligent Systems Applications to Power Systems*. Orlando, Florida, USA, 1996, pp. 321–326.

[37] Alvaro A. Cardenas, Saurabh Amin, and Shankar Sastry. "Secure Control: Towards Survivable Cyber-Physical Systems". In: *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops* (2008), pp. 495–500.

[38] Tianshi Chen et al. "A large population size can be unhelpful in evolutionary algorithms". In: *Theoretical Computer Science* 436 (2012), pp. 54–70.

[39] Nick Cheney et al. "Unshackling evolution: evolving soft robots with multiple materials and a powerful generative encoding". In: *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM. 2013, pp. 167–174.

[40] David Maxwell Chickering, David Heckerman, and Christopher Meek. "Large-sample learning of Bayesian networks is NP-hard". In: *Journal of Machine Learning Research* 5 (2004), pp. 1287–1330.

[41] Rudi Cilibrasi and Paul MB Vitányi. "Clustering by compression". In: *IEEE Transactions on Information theory* 51.4 (2005), pp. 1523–1545.

[42] International Electrotechnical Commission et al. *Fault tree analysis (FTA)*. International Electrotechnical Commission, 2006.

[43] Gregory F Cooper. "The computational complexity of probabilistic inference using Bayesian belief networks". In: *Artificial intelligence* 42.2-3 (1990), pp. 393–405.

[44] François Coste. "Learning the language of biological sequences". In: *Topics in Grammatical Inference*. Springer, 2016, pp. 215–247.

[45] Maxime Crochemore. "An optimal algorithm for computing the repetitions in a word". In: *Inf. Process. Lett.* 12.5 (1981), pp. 244–250.

[46] Wallace B Crowston, Fred Glover, Jack D Trawick, et al. *Probabilistic and parametric learning combinations of local job shop scheduling rules*. Tech. rep. Carnegie Inst of tech Pittsburgh PA Graduate School of Industrial Administration, 1963.

[47] Kalyanmoy Deb et al. "A fast and elitist multiobjective genetic algorithm: NSGA-II". In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197.

[48] Patricia Derler, Edward A Lee, and Alberto Sangiovanni Vincentelli. "Modeling cyber–physical systems". In: *Proceedings of the IEEE* 100.1 (2012), pp. 13–28.

[49] Rolf Drechsler and Bernd Becker. "Learning heuristics by genetic algorithms". In: *Proceedings of ASP-DAC'95/CHDL'95/VLSI'95 with EDA Technofair*. IEEE. 1995, pp. 349–352.

[50] Rolf Drechsler, Nicole Göckel, and Bernd Becker. "Learning heuristics for OBDD minimization by evolutionary algorithms". In: *International Conference on Parallel Problem Solving from Nature*. Springer. 1996, pp. 730–739.

[51] J. B. Dugan, S. J. Bavuso, and M. A. Boyd. "Fault trees and sequence dependencies". In: *Annual Proceedings on Reliability and Maintainability Symposium*. 1990, pp. 286–293.

[52] Pierre Dupont. "Regular grammatical inference from positive and negative samples by genetic search: the GIG method". In: *Grammatical Inference and Applications*. Springer Berlin Heidelberg, 1994, pp. 236–245.

[53] Hsiao-Lan Fang, Peter Ross, and David Corne. *A promising genetic algorithm approach to job-shop scheduling, rescheduling, and open-shop scheduling problems*. University of Edinburgh, Department of Artificial Intelligence, 1993.

[54] Henry Fisher. "Probabilistic learning combinations of local job-shop scheduling rules". In: *Industrial scheduling* (1963), pp. 225–251.

[55] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. "Combining model learning and model checking to analyze TCP implementations". In: *International Conference on Computer Aided Verification*. Springer. 2016, pp. 454–471.

[56] Peter Flach. *Machine learning: the art and science of algorithms that make sense of data*. Cambridge University Press, 2012.

[57] Emanuele Fumeo, Luca Oneto, and Davide Anguita. "Condition Based Maintenance in Railway Transportation Systems Based on Big Data Streaming Analysis". In: *Procedia Computer Science* 53 (2015), pp. 437–446.

[58] Pedro García and Enrique Vidal. "Inference of k-Testable Languages in the Strict Sense and Application to Syntactic Pattern Recognition". In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 12.9 (1990), pp. 920–925.

[59] Pedro Garcia, Enrique Vidal, and José Oncina. "Learning Locally Testable Languages in the Strict Sense." In: *Algorithmic Learning Theory (ALT), First International Workshop*. 1990, pp. 325–338.

[60] Pedro García et al. "Learning automata teams". In: *International Colloquium on Grammatical Inference*. Springer. 2010, pp. 52–65.

[61] Minos Garofalakis et al. "XTRACT: a system for extracting document type descriptors from XML documents". In: *ACM SIGMOD Record*. Vol. 29. 2. ACM. 2000, pp. 165–176.

[62] Christopher D. Geiger, Reha Uzsoy, and Haldun Aytuğ. "Rapid Modeling and Discovery of Priority Dispatching Rules: An Autonomous Learning Approach". In: *Journal of Scheduling* 9.1 (2006), pp. 7–34.

[63] Mark Gold. "Language identification in the limit". In: *Information Control* 10.5 (1967), pp. 447–474.

[64] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. "Learning of event-recording automata". In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer, 2004, pp. 379–395.

[65] Philip Gross et al. "Predicting electricity distribution feeder failures using machine learning susceptibility analysis". In: *Proceedings of the 21st National Conference on Artificial Intelligence*. Vol. 21. 2. Boston, Massachusetts, USA, 2006, pp. 1705–1711.

[66] Dan Gusfield and Jens Stoye. "Linear time algorithms for finding and representing all the tandem repeats in a string". In: *Journal of Computer System Sciences* 69.4 (2004), pp. 525–546.

[67] Emma Hart, Peter Ross, and Jeremy Nelson. "Solving a real-world problem using an evolving heuristically driven schedule builder". In: *Evolutionary Computation* 6.1 (1998), pp. 61–80.

[68] Hashem M. Hashemian and Wendell C. Bean. "State-of-the-Art Predictive Maintenance Techniques." In: *IEEE T. Instrumentation and Measurement* 60.10 (2011), pp. 3480–3492.

[69] JJ Henry and JD Andrews. "Computerized fault tree construction for a train braking system". In: *Quality and Reliability Engineering International* 13.5 (1997), pp. 299–309.

[70] Colin de la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[71] Gregory Hornby et al. "Automated antenna design with evolutionary algorithms". In: *Space 2006*. 2006, p. 7242.

[72] Jeffrey Jaffe. "A Necessary and Sufficient Pumping Lemma for Regular Languages". In: *SIGACT News* 10.2 (July 1978), pp. 48–49.

[73] A. Joshi et al. "Intelligent and learning-based approaches for health monitoring and fault diagnosis of RADARSAT-1 attitude control system". In: *2007 IEEE International Conference on Systems, Man and Cybernetics*. 2007, pp. 3177–3183.

[74] Sebastian Junges et al. "Fault trees on a diet: automated reduction by graph rewriting". In: *Formal Aspects of Computing* 29.4 (2017), pp. 651–703.

[75] Sohag Kabir. "An overview of fault tree analysis and its application in model based dependability analysis". In: *Expert Systems with Applications* 77 (2017), pp. 114–135.

[76] Kevin A Kaiser and Nagi Z Gebraeel. "Predictive maintenance management using sensor-based degradation models". In: *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on* 39.4 (2009), pp. 840–849.

[77] Michael Kearns, Ming Li, and Leslie Valiant. "Learning Boolean Formulas". In: *J. ACM* 41.6 (1994), pp. 1298–1328.

[78] S. K. Khaitan and J. D. McCalley. "Design Techniques and Applications of Cyberphysical Systems: A Survey". In: *IEEE Systems Journal* 9.2 (2015), pp. 350–365.

[79] Nima Khakzad, Faisal Khan, and Paul Amyotte. "Safety analysis in process facilities: Comparison of fault tree and Bayesian network approaches". In: *Reliability Engineering & System Safety* 96.8 (2011), pp. 925 –932.

[80] Kimmo Koskenniemi. *Two-level morphology: A general computational model for word-form recognition and production*. Vol. 11. University of Helsinki, Department of General Linguistics Helsinki, 1983.

[81] Kevin J Lang, Barak A Pearlmutter, and Rodney A Price. "Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm". In: *International Colloquium on Grammatical Inference*. Springer. 1998, pp. 1–12.

[82] Edward A Lee. "Cyber physical systems: Design challenges". In: *Proceedings of the 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing*. Orlando, Florida, USA, 2008, pp. 363–369.

[83] Wen-Shing Lee et al. "Fault Tree Analysis, Methods, and Applications: A Review". In: *IEEE transactions on reliability* 34.3 (1985), pp. 194–203.

[84] Joel Lehman and Risto Miikkulainen. *Neuroevolution*.

[85] Florian Leitner-Fischer and Stefan Leue. "Probabilistic fault tree synthesis using causality computation". In: *International Journal of Critical Computer-Based Systems 30* 4.2 (2013), pp. 119–143.

[86] Hongfei Li et al. "Improving rail network velocity: A machine learning approach to predictive maintenance". In: *Transportation Research Part C: Emerging Technologies* 45 (2014), pp. 17–26.

[87] Jing Li and Jianjun Shi. "Knowledge discovery from observational data for process control using causal Bayesian networks". In: *IIE transactions* 39.6 (2007), pp. 681–690.

[88] Shaojun Li and Xiaoxun Li. "Study on generation of fault trees from Altarica models". In: *Procedia Engineering* 80 (2014), pp. 140–152.

[89] Yue Li et al. "A method for constructing fault trees from AADL models". In: *Int. Conf. on Autonomic and Trusted Computing*. Springer. 2011, pp. 243–258.

[90] Peter Liggesmeyer and Martin Rothfelder. "Improving system reliability with automatic fault tree generation". In: *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*. 1998, pp. 90–99.

[91] Alexis Linard. "Learning Several Languages from Labeled Strings: State Merging and Evolutionary Approaches". In: *arXiv preprint arXiv:1806.01630* (2018).

[92] Alexis Linard and Marcos L. P. Bueno. "Towards Adaptive Scheduling of Maintenance for Cyber-Physical Systems". In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*. 2016, pp. 134–150.

[93] Alexis Linard, Colin de la Higuera, and Frits Vaandrager. "Learning Unions of k-Testable Languages". In: *Language and Automata Theory and Applications*. Ed. by Carlos Martín-Vide, Alexander Okhotin, and Dana Shapira. Cham: Springer International Publishing, 2019, pp. 328–339.

[94] Alexis Linard et al. "Learning Pairwise Disjoint Simple Languages from Positive Examples". In: *arXiv preprint arXiv:1706.01663* (2017).

[95] Alexis Linard et al. "Learning Pairwise Disjoint Simple Languages from Positive Examples". In: *ArXiv e-prints arXiv:1706.01663* (June 2017). arXiv: 1706.01663 [cs.LG].

[96] Michael G Madden and Paul J Nolan. "Generation of fault trees from simulated incipient fault case data". In: *WIT Trans. Information and Communication Technologies* 6 (1994).

[97] Alexander Maier, Oliver Niggemann, and Jens Eickmeyer. "On the Learning of Timing Behavior for Anomaly Detection in Cyber-Physical Production Systems". In: *Proceedings of the 26th International Workshop on Principles of Diagnosis*. Paris, France, 2015, pp. 217–224.

[98] Andreas Maier. "Online passive learning of timed automata for cyber-physical production systems". In: *Proceedings of the 12th IEEE International Conference on Industrial Informatics*. Porto Alegre, Brazil, 2014, pp. 60–66.

[99] Robert McNaughton and Seymour A. Papert. *Counter-Free Automata (M.I.T. Research Monograph No. 65)*. The MIT Press, 1971.

[100] Ayhan Mentes and Ismail H Helvacioglu. "An application of fuzzy fault tree analysis for spread mooring systems". In: *Ocean Engineering* 38.2-3 (2011), pp. 285–294.

[101] Li Ming and Paul Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer Heidelberg, 1997.

[102] S Montani et al. "A tool for automatically translating dynamic fault trees into dynamic Bayesian networks". In: *RAMS'06. Annual Reliability and Maintainability Symposium, 2006.* IEEE. 2006, pp. 434–441.

[103] Daniel Müllner. "Modern hierarchical, agglomerative clustering algorithms". In: *arXiv preprint arXiv:1109.2378* (2011).

[104] Kevin Patrick Murphy and Stuart Russell. "Dynamic bayesian networks: representation, inference and learning". In: (2002).

[105] Meike Nauta, Doina Bucur, and Mariëlle Stoelinga. "LIFT: Learning Fault Trees from Observational Data". In: *International Conference on Quantitative Evaluation of Systems*. Springer. 2018.

[106] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Berlin Heidelberg: Springer-Verlag, 1999.

[107] Oliver Niggemann et al. "Data-Driven Monitoring of Cyber-Physical Systems Leveraging on Big Data and the Internet-of-Things for Diagnosis and Control". In: *Proceedings of the 26th International Workshop on Principles of Diagnosis*. Paris, France, 2015, pp. 185–192.

[108] Sławomir Nowaczyk et al. "Towards a machine learning algorithm for predicting truck compressor failures using logged vehicle data". In: *Proceedings of the 12th Scandinavian Conference on Artificial Intelligence*. Aalborg, Denmark, 2013, pp. 205–214.

[109] Patrick O'Connor and Andre Kleyner. *Practical reliability engineering*. John Wiley & Sons, 2012.

[110] Arlindo L Oliveira and Alberto Sangiovanni-Vincentelli. "Learning complex boolean functions: Algorithms and applications". In: *Advances in Neural Information Processing Systems*. 1994, pp. 911–918.

[111] José Oncina and Pedro Garcia. "Identifying regular languages in polynomial time". In: *Advances in Structural and Syntactic Pattern Recognition* 5.99-108 (1992), pp. 15–20.

[112] Ender Özcan et al. "A reinforcement learning: great-deluge hyper-heuristic for examination timetabling". In: *Modeling, Analysis, and Applications in Metaheuristic Computing: Advancements and Trends*. IGI Global, 2012, pp. 34–55.

[113] Yiannis Papadopoulos and JA McDermid. "Safety-directed system monitoring using safety cases". PhD thesis. University of York, 2000.

[114] Myoung Soo Park and Jin Young Choi. "Logical evolution method for learning Boolean functions". In: *2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace*. Vol. 1. 2001, pp. 316–321.

[115] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., 1988.

[116] Fabian Pedregosa et al. "Scikit-learn: Machine learning in Python". In: *Journal of machine learning research* 12.Oct (2011), pp. 2825–2830.

[117] Joost van Pinxten et al. "Online Scheduling of 2-Re-entrant Flexible Manufacturing Systems". In: *ACM Trans. Embedded Comput. Syst.* 16.5 (2017), 160:1–160:20.

[118] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[119] Michael O Rabin and Dana Scott. "Finite automata and their decision problems". In: *IBM journal of research and development* 3.2 (1959), pp. 114–125.

[120] Ragunathan Rajkumar et al. "Cyber-physical systems: the next computing revolution". In: *Design Automation Conference*. IEEE. 2010, pp. 731–736.

[121] Marvin Rausand. *Risk assessment: theory, methods, and applications*. Vol. 115. John Wiley & Sons, 2013.

[122] James Rogers and Geoffrey K Pullum. "Aural pattern recognition experiments and the subregular hierarchy". In: *Journal of Logic, Language and Information* 20.3 (2011), pp. 329–342.

[123] Frank Rosenblatt. *Principles of neurodynamics. perceptrons and the theory of brain mechanisms*. Tech. rep. Cornell Aeronautical Lab Inc Buffalo NY, 1961.

[124] Hsiao-Lan Fang1and Peter Ross and Dave Corne. "A promising hybrid GA/heuristic approach for open-shop scheduling problems". In: *Proc. 11th European Conference on Artificial Intelligence*. 1994, pp. 590–594.

[125] Peter Ross et al. "Hyper-heuristics: learning to combine simple heuristics in bin-packing problems". In: *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc. 2002, pp. 942–948.

[126] Enno Ruijters and Mariëlle Stoelinga. "Fault Tree Analysis: A survey of the state-of-the-art in modeling, analysis and tools". In: *Computer Science Review* 15 (2015), pp. 29–62.

[127] Enno Ruijters and Mariëlle Stoelinga. "Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools". In: *Computer Science Review* 15-16 (2015), pp. 29 –62.

[128] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[129] Ken Sadohara. "Learning of boolean functions using support vector machines". In: *International Conference on Algorithmic Learning Theory*. Springer. 2001, pp. 106–118.

[130] Marcel-Paul Schützenberger. "On finite monoids having only trivial subgroups". In: *Information and Control* 8.2 (1965), pp. 190–194.

[131] Marco Scutari. "Learning Bayesian Networks with the bnlearn R Package". In: *Journal of Statistical Software* 35.i03 (2010).

[132] Septavera Sharvia et al. "Model-based dependability analysis: State-of-the-art, challenges, and future outlook". In: *Software Quality Assurance*. Elsevier, 2016, pp. 251–278.

[133] Zineb Simeu-Abazi and Zouhir Bouredji. "Monitoring and Predictive Maintenance: Modeling and Analyse of Fault Latency". In: *Comput. Ind.* 57.6 (Aug. 2006), pp. 504–515.

[134] Dan Simon. *Evolutionary optimization algorithms*. John Wiley & Sons, 2013.

[135] Michael Stamatelatos et al. *Fault tree handbook with aerospace applications*. nasa Washington, DC, 2002.

[136] Jens Stoye and Dan Gusfield. "Simple and flexible detection of contiguous repeats using a suffix tree". In: *Theoretical Computer Science* 270.1-2 (2002), pp. 843–856.

[137] L. Swartjes et al. "Simultaneous analysis and design based optimization for paper path and timing design of a high-volume printer". In: *Mechatronics* 41 (2017), pp. 82 –89.

[138] Frédéric Tantini, Alain Terlutte, and Fabien Torre. "Sequences classification by least general generalisations". In: *International Colloquium on Grammatical Inference*. Springer. 2010, pp. 189–202.

[139] Inés Torres and Amparo Varona. "k-TSS language models in speech recognition systems". In: *Computer Speech & Language* 15.2 (2001), pp. 127–148.

[140] Alan M Turing. "Computing machinery and intelligence". In: *Parsing the Turing Test*. Springer, 2009, pp. 23–65.

[141] Waqas Umar et al. "A Fast Estimator of Performance with Respect to the Design Parameters of Self Re-Entrant Flowshops". In: *Euromicro Conference on Digital System Design*. 2016, pp. 215–221.

[142] Sicco Verwer, Mathijs Weerdt, and Cees Witteveen. "Efficiently identifying deterministic real-time automata from labeled data". In: *Machine Learning* 86.3 (2011), pp. 295–333.

[143] William E Vesely et al. *Fault tree handbook*. Tech. rep. Nuclear Regulatory Commission Washington dc, 1981.

[144] James D Walker et al. "Vehicle routing and adaptive iterated local search within the hyflex hyper-heuristic framework". In: *Learning and Intelligent Optimization*. Springer, 2012, pp. 265–276.

[145] U. Waqas et al. "A Fast Estimator of Performance with Respect to the Design Parameters of Self Re-Entrant Flowshops". In: *2016 Euromicro Conference on Digital System Design (DSD)*. 2016, pp. 215–221.

[146] U. Waqas et al. "A re-entrant flowshop heuristic for online scheduling of the paper path in a large scale printer". English. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 15) 9-13 March 2015, Grenoble, France*. United States: Institute of Electrical and Electronics Engineers (IEEE), 2015, pp. 573–578.

[147] Marcel Wever, Lorijn van Rooijen, and Heiko Hamann. "Active Coevolutionary Learning of Requirements Specifications from Examples". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '17. Berlin, Germany: ACM, 2017, pp. 1327–1334.

[148] Ian H Witten et al. *Weka: Practical machine learning tools and techniques with Java implementations*. 1999.

[149] Takashi Yokomori and Satoshi Kobayashi. "Learning Local Languages and Their Application to DNA Sequence Analysis". In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 10 (1998), pp. 1067–1079.

[150] Yanhua Zhang et al. "A method of fault tree generation based on go model". In: *Reliability Systems Engineering (ICRSE), 2015 First International Conference on*. IEEE. 2015, pp. 1–5.

[151] Yihuan Zhang et al. "Car-following Behavior Model Learning Using Timed Automata". In: *IFAC-PapersOnLine* 50.1 (2017). 20th IFAC World Congress, pp. 2353 –2358.

Summary

The complexity of industrial systems, among others, Cyber-Physical Systems, has been continuously increasing over the last decades. In order to maintain scheduling and control of these systems feasible, many techniques can be engaged. In particular, machine learning, grammatical inference, model checking and reliability engineering are popular methods widely used in industry to overcome problems such as systems efficiency, reliability and safety. In this thesis, we focus on learning a variety of models from data. Interestingly, we focused on automata, Fault Trees and tree-like Ranking Metrics, and show that we can dispense with the popular neural networks.

This thesis presents a set of contributions to improve:

1. The scheduling and control of maintenance for Cyber-Physical Systems (Chapter 2). Nowadays, engineers are interested in accurate controllers to detect component failures and methods to plan for their replacements efficiently. We improved maintenance strategies for such a complex embedded system, monitoring the critical components in real-time and dynamically adjusting the optimal time between maintenance actions. To that end, we combined machine learning and model checking to perform maintenance smartly. In our case study, we learnt a model describing the failure behaviour of a Large-Scale Printer. We used this model to reduce maintenance costs and increase the productivity of the system.

2. The learning of reliability models, namely Fault Trees, for describing the failure behaviour of components of Cyber-Physical Systems (Chapters 3 and 4). With the explosion of the number of components systems contain, obtaining accurate systems reliability models becomes a hard task. Therefore, we used machine learning and Bayesian inference for the learning of Fault Trees from data about components failure behaviour. Combined with expert knowledge, learning of Fault Trees for Large-Scale Printers is efficient and particularly accurate.

3. The learning of models for Cyber-Physical Systems, in particular in the context of reverse engineering (Chapters 5 and 6). Learning of models (e.g. automata) from data comprises techniques for the inference of formal languages describing the implemented model. When this model appears to be representing different usages of the system, the problem turns to be to identify several languages (e.g. describing different user behaviours). To that end, we defined a new framework for learning unions of languages. In our case study, we focused on the identification of print job patterns submitted to Large-Scale Printers.

4. The scheduling of processes, by improving a heuristic scheduler of print jobs (Chapter 7). Large-Scale Printers requires on-line scheduling to satisfy timing constraints due to complex interactions between modules. Among others, ranking metrics determine for each sheet when it should enter the system, be printed for the first time, and when it should return for its second print. We used machine learning, in particular, an evolutionary algorithm mimicking biological evolution, to heuristically find good ranking metrics. Combined with the previously discovered print job patterns, our method happens to increase the overall productivity of Large-Scale Printers.

Samenvatting

De complexiteit van industriële systemen, waaronder Cyber-Fysische Systemen, is de afgelopen decennia voortdurend toegenomen. Om het plannen en besturen van deze systemen mogelijk te houden, kunnen veel technieken worden ingezet. Machine learning, grammatical inference, model checking en reliability engineering zijn populaire methoden die veel in de industrie toegepast worden om problemen op te lossen, zoals productiviteit van systemen, betrouwbaarheid en veiligheid.

Dit proefschrift presenteert een aantal bijdragen voor het verbeteren van:

1. De planning en controle van onderhoud voor Cyber-Fysische Systemen (Hoofdstuk 2). Tegenwoordig zijn ingenieurs geïnteresseerd in nauwkeurige controllers om defecten van componenten te detecteren, en methoden om hun vervangen efficiënt in te plannen. We hebben onderhoudsstrategieën voor zo'n complex ingebed systeem verbeterd, zodat die in realtime de kritieke componenten te monitoren en de optimale tijd tussen onderhoudsacties dynamisch aanpast. Daartoe combineerden we machine learning en model checking om onderhoud slim uit te voeren. In onze casestudy hebben we een model kunnen leren dat het faalgedrag van een grootschalige printer beschrijft. We hebben dit model gebruikt om tegelijk de onderhoudskosten te verlagen en de productiviteit van het systeem te verhogen.

2. Het leren van betrouwbaarheidsmodellen, namelijk foutenbomen, voor het beschrijven van het faalgedrag van componenten van Cyber-Fysische Systemen (Hoofdstukken 3 en 4). Met de explosie van het aantal componenten dat systemen bevatten, wordt het verkrijgen van accurate betrouwbaarheidsmodellen voor systemen een moeilijke taak. Daarom, hebben we machine learning en Bayesian inference gebruikt om foutenbomen te leren uit data over het gedrag van onderdelendefecten. Gecombineerd met kennis van experts, is het aanleren van foutenbomen voor grootschalige printers efficiënt en bijzonder nauwkeurig gebleken.

3. Het leren van modellen voor Cyber-Fysische Systemen, met name het kader van reverse engineering (Hoofdstukken 5 en 6). Het leren van automaten uit data omvat technieken voor de inferentie van formele talen die het geïmplementeerde model beschrijven. Wanneer dit model verschillende gebruiksmogelijkheden van het systeem lijkt weer te geven, wordt dan het probleem om meerdere talen te identificeren (bijvoorbeeld het beschrijven van verschillende gebruikersgedragingen). Daartoe hebben we een nieuw kader gedefinieerd voor het leren van verenigingen van talen. In onze casestudy hebben we ons gericht op de identificatie van printjobpatronen die zijn ingediend bij grootschalige printers.

4. Het plannen van processen door een heuristische scheduler van printjobs te verbeteren (Hoofdstuk 7). Voor grootschalige printers is online scheduling vereist om de timingbeperkingen van complexe interacties tussen modules te voldoen. Onder andere, rangordestatistieken voor elk blad geretourneerd worden. Zij decideren wanneer bladen in het systeem moet ingevoerd worden, en wanneer voor de eerste en tweede keer moeten zij afgedrukt worden. We gebruikten machine learning, in het bijzonder een evolutionair algoritme dat de biologische evolutie nabootst, om op heuristische wijze goede rangordestatistieken te vinden. Gecombineerd met de eerder ontdekte printjob-patronen, onze methode verhoogde de algehele productiviteit van grootschalige printers.

Résumé

La complexité des systèmes industriels et des systèmes cyber-physiques actuels croît d'année en année. Afin de permettre la programmation et le contrôle de ces systèmes, plusieurs techniques peuvent être employées. Entre autres, l'apprentissage automatique, l'inférence grammaticale, la vérification de modèles et l'ingénierie de la fiabilité sont des méthodes largement répandues dans l'industrie pour répondre aux problématiques de productivité, de fiabilité et de sécurité des systèmes. Dans cette thèse, nous nous intéressons à l'apprentissage d'une multitude de modèles à partir de données. Nous nous penchons tout particulièrement sur les automates, les arbres de défaillances et les métriques d'ordonnancement. Nous montrons que curieusement, nous pouvons nous passer des réseaux de neurones si populaires de nos jours. Nos travaux ont été inspirés par un cas d'étude bien précis : les imprimantes grande échelle, capables de reproduire cette présente thèse en moins d'une minute.

Cette thèse présente une série de contributions visant à améliorer:

1. La programmation et le contrôle de la maintenance des systèmes cyber-physiques (Chapitre 2). De nos jours, les ingénieurs s'intéressent à des contrôleurs capables de détécter les défaillances de composants, ainsi qu'à des méthodes permettant de remplacer les-dits composants de façon efficace. Nous avons amélioré les statégies de maintenance pour de tels systèmes embarqués complexes, en monitorant en temps réel les composants critiques, et en ajustant de façon dynamique le temps optimal entre deux opérations de maintenance. Pour cela, nous avons combiné l'apprentissage automatique et la vérification de modèles pour déclencher les opérations de maintenance de façon plus intelligente. Dans notre cas d'étude, nous avons pu apprendre un modèle décrivant comment les composants d'une imprimante grande échelle se comportent, et comment ceux-ci tombent en panne. Nous avons utilisé ce modèle pour réduire les coûts de maintenance ainsi qu'accroître la productivité du système.

2. L'apprentissage de modèles de fiabilité, entre autres, les arbres de défaillance, pour l'apprentissage de la survenue de pannes des composants des systèmes cyber-physiques (Chapitres 3 et 4). Avec l'explosion du nombre de composants que les systèmes contiennent, l'obtention de modèles de fiabilité de bonne qualité devient une tâche ardue. Par conséquent, nous avons employé des techniques d'apprentissage automatique et d'inférence Bayésienne pour générer automatiquement des arbres de défaillance à partir de données sur les composants. Combiné à l'expertise des ingénieurs, l'apprentissage d'arbres de défaillances pour les imprimantes grande échelle est efficace et précis.

3. L'apprentissage de modèles pour les systèmes cyber-physiques, en particulier dans le contexte de la rétroingénierie (Chapitres 5 et 6). L'apprentissage de modèles à partir de données, comme par exemple les automates, comprennent des techniques pour l'inférence de langages formels, décrivant un modèle implémenté. Lorsqu'il se trouve que ce modèle représente différents usages d'un système, le problème en question est donc de devoir identifier plusieurs langages (c'est-à-dire, ceux décrivant les différents comportements d'utilisateurs). À cette fin, nous avons développé plusieurs approches pour l'inférence d'unions de langages. Dans notre cas d'étude, nous nous sommes penchés sur l'identification de patrons de tâches d'impression envoyés à des imprimantes grande échelle.

4. L'ordonnancement de tâches, en améliorant un planificateur de tâches d'impression basé sur des heuristiques (Chapitre 7). Les imprimantes large échelle ont besoin de la planification en ligne pour satisfaire les contraintes temporelles dues aux interactions complexes entre les différents modules de ce Système Cyber-Physique particulier. Entre autres, les métriques d'ordonnancement déterminent, pour chaque page devant être imprimée, à quel moment elle doit entrer dans le système. Elle décide également à quel moment elle doit être imprimée pour la première fois, ou encore à quel moment elle doit être retournée lorsqu'il s'agit d'impressions recto verso. Nous avons utilisé des techniques d'apprentissage automatique, en particulier un algorithme évolutif reproduisant l'évolution biologique, afin de trouver heuristiquement de bonnes métriques d'ordonnancement. Combinée aux patrons de tâches d'impression découverts précédemment, notre méthode a amélioré la productivité de ces imprimantes grande échelle.

Acknowledgements

Doing a PhD has been a fantastic experience for me. Thanks to people surrounding me, all this became possible.
Frits, thank you for having given me this great opportunity to do this PhD under your supervision. I will not forget all the excellent advice you gave me all the time, and also the good discussions during our daily coffee breaks. I also learnt some useful lessons from you, especially how to remain ethical and humble in academia.
I would also like to thank all the members of the committee for reading the thesis, providing useful feedback, and for their presence at the defence. Thanks to Herman for presiding the committee. Thanks to Alessandro, for the very interesting and fruitful discussions we had during the Lorentz workshop. Thanks to Colin, for my past and present working life, and to Jana, for my present and future one. Thanks to Twan, for all the interesting (and often tough) questions during our bi-weekly meetings at Océ.
All this work would not have been possible without the collaboration of many people from both academia and industry. Therefore, I would like to acknowledge all my co-workers at Océ: Ali, Amir, Joost, Lou, Marc, Patrick and Waqas, whose help and discussions have been fruitful. I would like to acknowledge all my co-authors for all productive discussions and nice work. Thanks to Colin, Doina, Joost, Marcos, Mariëlle, Rick, Sicco and Waqas for the papers we wrote together.
I would also like to acknowledge all the people from the Radboud's side, for all the good moments we had together: Gabriel, Harco, Henning, Ingrid, Jacopo, Jan, Joshua, Jurriaan, Manxia, Marcos, Marcus, Mariëlle, Michele, Niels, Nils, Paul, Paulus, Petra, Ramon, Rick, Simone L, Simone M and Tim. Joshua, thanks for these 4 years as office (and above all *proeverij*) mate! I will never forget our visits to fancy *lands*... also thanks to Tessa for also being on board in many of these crazy expeditions. By the way, I probably owe to the 2 of you my practice of the Dutch language. Paul and Michele, thanks also for all the social life I had with you as well. Paul, I'm sure that we will keep on having fun in Uppsala/Stockholm! Nils, I will certainly keep in mind our discussions about tough academic life. And of course, I owe you a lot concerning my career plans, you know what I mean ;) .
Jurriaan, thank you for letting me teach some of Talen&Automaten: you probably don't realize how much I enjoyed this experience and how it enlightened me on what I wanted to do after my PhD! Simone, thanks for all the good time in Utrecht, Nijmegen, and during 4daagse. And also for the *"affair"*, actually I think you did most of the work! Ingrid, how can I thank you enough for all what you did, finding a house, and all your kindness? Many thanks for having made my last 4 years nicer.
About lovely house and neighbours, I also take this opportunity to thank Tilly and Sjors, for your kindness and the good time we spent in Spain, France, and the Netherlands, as well as all the others of the GBW 164, Elze, Anouk, Daan, Andreas and Former. Thank you Elze for letting me crash at your place a couple of times!

Thanks to all my friends in France, Yolène, Charlène & Jérémy, Maryne & Aurélien, Nathalie, Sophie, Paul, Françoise, Martine, Elizabeth, for all the visits you paid me in the Netherlands, and all the good time. Yolène, our life stories often provided me with good pretexts to escape the paper deadlines or the (sometimes stressful) PhD life!
I also want to thank all my family for all the good moments. Especially grandma, aunts, uncles and cousins for having visited me in Nijmegen. Flavie, sorry for having moved away from the Netherlands: you won't have *kapsalons* anymore!
Finally, I want to say a couple of words to the 2 people to whom I dedicate this thesis. You are always the last in the acknowledgements, but for sure, not the least. Thanks for your support, all the good moments, and all your affection. I owe you so much.

Merci !

Curriculum Vitae

Alexis Linard

1993 Born on February 23 in Troyes, France

2010–2012 University Degree in Computer Science
Institute of Technology, University of Nantes, France

2012–2013 BSc in Computer Science
Institute of Technology, University of Nantes, France

2013–2015 MSc in Computer Science
University of Nantes, France

2015–2019 PhD in Computer Science
Radboud University, Nijmegen, the Netherlands