DOCTOR OF SCIENCES

Learning Featured Transition Systems

Fortz, Sophie

Award date:
2023

Awarding institution:
University of Namur

Learning Featured Transition Systems

Sophie Fortz

Jury

Prof. Benoît Frénay
University of Namur, Belgium

Prof. Patrick Heymans
University of Namur, Belgium

Prof. Mohammad Reza Mousavi
King's College London, UK

Dr. Gilles Perrouin
University of Namur, Belgium

Dr. Maurice ter Beek
ISTI CNR, Italy

Prof. Wim Vanhoof
University of Namur, Belgium

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in the subject of Computer Science

Supervised by Dr. Gilles Perrouin and Prof. Patrick Heymans

University of Namur
NaDi Institute
PReCISE Research Center

Cover design: © Presses universitaires de Namur

Rue Grandgagnage 19
B – 5000 Namur (Belgium)
pun@unamur.be - www.pun.be

Registration of copyright: D/2023/1881/13
ISBN: 978-2-39029-176-3
Printed in Belgium.

"We often focus so much on the tiny details we don't like that we forget to step back and look at the bigger picture. Little mistakes get lost in the details and nobody else is looking as closely as you, so instead of getting caught up in perfection, enjoy the process a little more and don't let these little moments ruin your enjoyment of the rest of your piece."

— ***Sarah Renae Clark***

ABSTRACT

Variability-intensive Systems (VISs) are software-based systems whose characteristics and behaviour can be modified by the activation or deactivation of some options. Addressing variability proactively during software engineering (SE) activities means shifting from reasoning on individual systems to **reasoning on families of systems**. Adopting appropriate variability management techniques can yield important **economies of scale** and quality improvements. Conversely, variability can also be a curse, especially for Quality Assurance (QA), *i.e.*, **verification and testing** of such systems, due to the combinatorial explosion of the number of software variants. Indeed, by combining only 33 Boolean options, we can define more variants of a system than the number of people on Earth. Verifying or testing each variant individually is thus impossible in most practical cases.

About a decade ago, **Featured Transition Systems (FTSs)** were introduced as a formalism to represent, and reason on, **the behaviour of VISs**. Instead of representing each variant by a (classical) transition system, an FTS bears annotations that relate transitions to options through **feature expressions**. FTSs thus make it possible to reason at the family level by modelling all the variants of a system in a single behavioural model. FTSs have been shown to significantly improve the possibilities and execution time of automated QA activities such as model-checking and model-based testing. They have also shown their usefulness to guide design exploration activities. Yet, as most model-based approaches, FTS modelling requires both **strong human expertise and significant effort** that would be unaffordable in many cases, in particular for large legacy systems with outdated specifications and/or systems that evolve continuously.

Therefore, this thesis aims **to automatically learn FTSs from existing artefacts**, to ease the burden of modelling FTS and support continuous QA activities. To answer this research challenge, we propose a two-phase approach. First, we rely on deep learning techniques to locate variability from execution traces. For this purpose, we implemented a tool called **VaryMinions**. Then, we use these annotated traces to learn an FTS. In this second part, we adapt the seminal L^* algorithm to **learn behavioural variability**. Both frameworks are open-source and we evaluated them separately on several datasets of different sizes and origins (*e.g.*, software product lines and configurable business processes).

Keywords: Variability-intensive Systems, Software Product Line, Featured Transition Systems, Reverse Engineering, Active Automata Learning, Variability Mining

RÉSUMÉ

Les Systèmes hautement configurables (SHC) sont des systèmes logiciels dont les caractéristiques et le comportement peuvent être modifiés par l'activation ou la désactivation de certaines options. Aborder la variabilité de manière proactive lors des activités de génie logiciel (GL) signifie passer d'un raisonement basé sur un seul système, à un raisonnement au niveau d'une **famille de systèmes**. Adopter les bonnes techniques de gestion de la variabilité entraînent d'importantes **économies d'échelle** et améliorent la qualité du système. À l'inverse, la variabilité peut également être une malédiction, en particulier pour l'Assurance Qualité (AQ), c'est-à-dire **la vérification et le test** de tels systèmes, en raison de l'explosion combinatoire du nombre de variants logiciels. En effet, en combinant seulement 33 options booléennes, on peut définir plus de variants d'un système que le nombre de personne sur Terre. Vérifier ou tester chaque variant individuellement est donc impossible dans la grande majorité des cas pratiques.

Il y a plus d'une dizaine d'années, les **Featured Transition Systems (FTSs)** ont été introduits comme un formalisme pour représenter et raisonner sur **le comportement des SHC**. Au lieu de représenter chaque variant par un système de transition classique, un FTS est doté d'annotations qui relient les transitions aux options via des **feature expressions**. Les FTSs rendent ainsi possible la réflexion au niveau de la famille en modélisant tous les variants d'un système dans un seul modèle comportemental. Les FTSs ont montré qu'ils améliorent considérablement les possibilités et le temps d'exécution des activités automatisées d'assurance qualité telles que la vérification et les tests basés sur les modèles. Ils ont également montré leur utilité pour guider *les activités d'exploration de conception*. Cependant, comme la plupart des approches basées sur les modèles, la modélisation de FTSs nécessite à la fois **une grande expertise humaine et un effort significatif** qui serait prohibitif dans de nombreux cas, en particulier pour les grands systèmes avec des spécifications obsolètes et/ou des systèmes qui évoluent en continu.

Par conséquent, cette thèse vise à **apprendre automatiquement des FTSs à partir d'artefacts existants**, afin de faciliter la modélisation et de soutenir les activités d'AQ continue. Pour répondre à ce challenge, nous proposons une approche en deux phases. Tout d'abord, nous nous appuyons sur des techniques d'apprentissage profond pour localiser la variabilité à partir de traces d'exécution. À cette fin, nous avons implémenté un outil appelé VaryMinions. Ensuite, nous utilisons ces traces annotées pour apprendre un FTS. Dans cette deuxième partie, nous adaptons l'algorithme fondateur L^* pour apprendre la variabilité comportementale. Les deux frameworks

sont open-sources et ont été évalués séparément sur plusieurs ensembles de données de tailles et d'origines différentes (par exemple, des lignes de produits logiciels et des processus business configurables).

Mots clés: Systèmes hautement configurables, Lignes de produits logiciels, featured transition systems, rétro-ingénierie, Apprentissage actif d'automates, extraction de variabilité

ACKNOWLEDGEMENTS

The journey presented in this thesis has been filled with both challenges and triumphs. Without the invaluable assistance and unwavering support of many remarkable individuals, none of this would have been possible.

I was lucky enough to be supervised by two incredible people. First, Dr. Gilles Perrouin who has been following me day by day for the last four years. We did not make your task easy by being your first official PhD students, working on quite different subjects and keeping such close schedules. I did not start my PhD in the best conditions (**cough 2020 cough**), but I could not have dreamed of a better supervisor help me face adversities. I remember the discussions I had with other master's students when we talked about the qualities we sought for a "good" supervisor. I think you checked all the boxes! I'm proud of what I've accomplished and what I learned through you. Then, Prof. Patrick Heymans, who believed in me from the start by asking me to join the EOS project. Patrick, your comments during the mid-term exam and the defence were always constructive and encouraging. Thank you for pushing me until I successfully got my FRIA grant and for publishing in this small workshop the first summer. Your ears must have been burning a little, but I'm now proud of this paper which is one of the two contributions of this manuscript.

My deepest gratitude goes to my accompanying committee and jury members for their valuable insights: Prof. Benoît Frénay, Prof. Mohammad Reza Mousavi, Dr. Maurice Ter Beek and Prof. Wim Vanhoof. It was a pleasure to meet you, and I learned a lot from you. You were always constructive, pedagogical and caring. Benoît, you and your team introduced me to the vast field of machine learning. You were always there to discuss the links between software engineering and machine learning. Mohammad, your papers were always such a great inspiration! Thank you for giving me the opportunity to continue our collaboration in a new, inspiring environment. Maurice, our discussions during my midterm exam were really helpful and gave me the confidence that we were on the right track. Wim, you have been following me since I was a young master student, and working with you gave me the impulse to pursue in research.

During these four years, I had the chance to collaborate with several people. I would like to thank all my co-authors and those who supported me from the start. There are three people that come to my mind for their daily support. First, Dr. Paul Temple, whom I shared the office with for three years. Paul, you were there at every step of the way, since my first meetings with Gilles. I want to apologise for, sometimes, using you as a rubber duck (even if I'm still convinced that you

understood more than you think). You never failed to reassure me whenever I doubted (even when I was completely rambling), and despite the distance. I would also like to thank Prof. Xavier Devroey for guiding me through academic life. Your honest comments have always been a great help to focus on the important matters. I learned a lot from you about paper writing, ideas presentation and how academia works. Our coffee break discussions help me find the right balance between research and personal life. Especially during this last year, Dr. Moussa Amrani was here to help me face the complexity of automata learning. I really enjoyed our discussions about research and life as a postdoc.

I want to add a special mention to Prof. Pierre-Yves Schobbens, who gave us the idea to use BDD at a time when I felt completely stuck. Thanks to you, I was able to propose my algorithm (almost) on time! Besides research, I also had the opportunity to teach. I am grateful to Prof. Marie-Ange Remiche and Prof. Martine de Vleeschouwer, who gave me this opportunity.

The friendships and camaraderie within the Computer Science Faculty have been a source of strength and joy. There are many people I want to thank for their continuous support and memorable moments throughout this journey. First, my co-workers from the 432th: James, Edilton and Antoine. You were always there to discuss research, share good advice or join me for a beer! Then, all my other friends I met at the faculty: Manel, Antoine C., Maxime C., Jérôme F., Benoît V., Charline, Maxime A., Claire, Valentin, Guillaume N., Martin, Pierre, Sacha, Mohammed, ... I probably forgot a lot of people, but as of the date I'm writing these acknowledgements, our lunch team includes 62 people!

The Computer Science Faculty is lucky to have an incredible secretary team. More particularly, I'd like to thank Isabelle for always being there to help and Babette, the backbone of our faculty and a friend always ready to listen to my complaints.

Beyond the academic sphere, I am very lucky to have very patient friends, who tolerate that I do not contact them during a long period of rush, and then welcome me with open arms to celebrate. Thank you Julie, Hélène and Bastien, Meghan, Viviane, Marie W., Alice B. and Anthony, Olivier W. and all the others. Your unwavering support means the world to me. During a very long period of my life, I was an active member of the Belgian Girl Guide association. I met a lot of friends there that I want to thank for their continuous support. Being able to pursue fundamental research while being involved in concrete projects helps me keep my feet on the ground. For all these years, thank you to Florence and the staff, Robert, Marianne, Virginie, Barbara, Nolwenn, and all the Mozet team. I would like to add another special thanks to Emilia. You provided me a bubble every Saturday for the last 17 years! It always helped me to face life's torments.

To conclude, I would not be the woman I am today without the unconditional support of my beloved family. Thanks to my godfather and godmother, my aunts, uncles and cousins. With my whole heart, I thank my grandmothers for their love. Florine, I could not be prouder to be your godmother. Being a big sister means always being there to cheer the (not so) little ones up. But these last few years, the student has surpassed the master. Being the only one left at home was not always easy for you, but François, you cannot know how much it feels to come home and

get a hug from you! Nicolas, do you remember our bike rides when we were little, just the two of us? Today, when I spend time with you, I am still that proud little girl. I will always cherish these memories. Olivier, this year will bring a lot of change for you too. I hope that you will always find a path where you can bring your unique touch and be yourself. I am eager because I know that you will surprise us one more time! Benoît, I am so proud of you! You are my complementary sidekick. Don't forget that I'm there for you, even if a sea separates us. I am just one train away from you, either for celebrating or comforting. No matter what I felt, the four of you were always there to suggest a board game or a baseball bat! And finally, Papa, Maman, it is not possible to put into words what I'm feeling right now. You made me the person I am today. My values, my accomplishments, my perseverance... It is all thanks to you! You inspired me in more ways than I could imagine. So I will keep it simple by just reminding you how much I love you.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

Reuse has been a common practice in Software Engineering (SE) since the 1970s (see Figure 1). About twenty years later, practitioners began to not only reusing existing code and design, but also developing new software with variability in mind. This led to the formalisation of the concept **Variability-Intensive Systems (VISs)**. A VIS can be viewed as a family of systems with different specificities, while sharing a common purpose. Each individual member of this family is referred to as a **variant** or a **product**, which is defined by a combination of **features** (i.e., specific characteristics). VISs encompass a wide range of applications, including Software Product Lines (SPLs), configurable systems or adaptive systems.

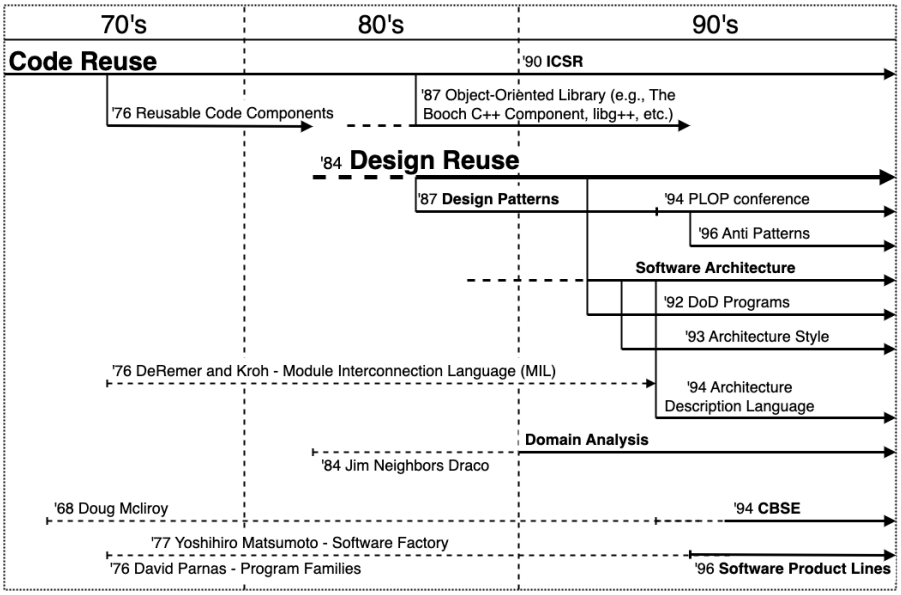

Figure 1: Evolution of Reuse Concepts, by Kyo Chul Kang (SPLC Jubilee Celebration) [129]

For Quality Assurance (QA) activities, such as verification and validation, one of the challenges posed by VIS lies the combinatorial explosion of variants. Even with a relatively modest number of Boolean features (*i.e.,* options that can be either activated or deactivated), such as 33, the number of possible variants exceeds the

population of the Earth. With 320 features, the number of variants surpasses the number of atoms in the universe. To provide a comparison, the Linux Kernel counts approximately 7000 distinct features, which are not limited to Boolean values but can have different types or attributes. When considering the verification and validation of each product configuration individually, tackling the problem becomes practically infeasible. One solution to mitigate the impact of combinatorial explosion is to reason on *family* models. Model-based approaches facilitate the automation of numerous QA tasks, offering a viable approach to address this challenge.

Feature Modelling [45, 53, 128, 159, 185, 186] is the most popular approach to model **structural variability**, *i.e.,* modelling the feature and constraints describing valid products. Feature models (FMs) are a compact representation, taking the graphic form of a tree structure. They are used for a large variety of automated tasks, such as variant counting or variant sampling.

On the other hand, several approaches have been defined to model the **behaviour** of VISs. Classen *et al.* [46, 48, 51] defined **Featured Transition Systems (FTSs)** as a way to represent an entire SPL in a compact way. FTSs represent the behaviour of an SPL by a Labelled Transition System [20, 81], whose labels are called Feature expressions. **Feature Expressions** relate structural variability (*i.e.,* SPL features) to behaviour, specifying which subset of products is able to execute each transition. Thus, FTSs take advantage of **shared behaviour** and allow substantial scale economies to perform analysis tasks, such as testing and model checking.

Context and Problem Statement

Modelling the behaviour of VISs is essential for their verification and validation. Yet, VIS are rarely shipped with such models. Engineers usually provide these models by hand, which is time-consuming, error-prone and does not scale to complex VISs. Current inference approaches [54, 56, 63, 201] either suffer from scalability issues (enumerative, variant-based learning) or lack of automation (manual feature annotation). These existing approaches often overlook the key strength of variability models, which is their capability to express shared behaviour among different products. Hence, the main objective of this research is to tackle these limitations and automate the creation of FTSs, to alleviate the burden associated with modelling FTSs and enhance continuous Quality Assurance (QA) activities. By harnessing the power of automated learning techniques, this research aims to unleash the full potential of variability models in expressing shared behaviour. This, in turn, will enable more streamlined and effective verification and validation processes, leading to improved software quality.

To address the current **automation** and **scalability** issues identified in the state-of-the-art, we take commonalities between variants into account right from the early stages of learning. As FTS is a fundamental formalism that can serve as a semantics for other VIS modelling languages such as UML State Diagrams (*e.g.,* via flattening [60]), the results are intended to **be generic and therefore have a profound impact on behavioural inference and automation**. To answer this research question, we propose a three-step approach.

In Part II, we first **formalise** some concepts such as FM, FTS and a new family of models, inspired by FTS, that we called Featured Deterministic Finite Automaton (FDFA). Then, **we adapt the Angluin's L* learning algorithm** [9] to learn the behaviour of systems in a family-based and symbolic manner, **treating feature expressions as first-class citizen**. The L^* algorithm follows a simple metaphor where a Learner component constructs a model iteratively. The Learner make queries to another component known as the Teacher, which acts as a proxy for the system we want to model. If the learned model is incorrect, the Teacher provides counterexamples to guide the learning process.

Learning the VIS in a family-based fashion requires to relate Angluin's queries and counterexamples to configurations. However, since the Teacher only knows about previously observed variants, the existing mappings are incomplete as they rely on partial observations of the system. Consequently, any new configurations that arise are considered unknown to the Teacher. This discrepancy necessitates the development of a configuration prediction technique capable of bridging this gap effectively. By accurately associating queries and counterexamples with their respective configurations, we can overcome the limitations posed by partial observations and enable a comprehensive family-based learning approach for the VIS. Being able to locate variations is also an essential part of any re-engineering endeavour [15] and is naturally useful for testing techniques, notably to sample which variants should be tested [107]. Existing variant analysis [204] techniques rather focus on the differences between identified variants than identifying which variant(s) may have produced a given trace.

To address these challenges, in Part III, we **leverage deep learning techniques to develop an efficient approach for variability localisation**. By harnessing the power of deep learning, we aim to enhance the capability of accurately identifying and localising variations within the VIS, thus contributing to improved understanding and testing of software variants. For this task, we tailor two architectures of Recurrent Neural Networks (RNNs): Long Short Term Memory (LSTMs) [119] and Gated Recurrent Units (GRUs) [41]. These kinds of models are well-known in other contexts such as **Natural Language Processing (NLP)**, to deal with long input sequences.

In order to focus on the behavioural aspects, **we assume that the FM either already exists or has been learned in some way**(*e.g.,* [4,5,145,152,155,191]). Indeed, several ways to learn a FM have been studied by the community, depending on the source of information available.

Contributions

Research Questions. This manuscript aims to answer the following research questions:

RQ$_1$ How to automatically learn FTSs in order to ease the burden of modelling FTSs and support continuous VIS QA activities?

RQ$_{1.1}$ What amount of time does a variability-aware learning approach requires?

RQ$_{1.2}$ Does variability-aware learning lead to fewer membership queries than the state-of-the-art approaches?

RQ$_{1.3}$ Does variability-aware learning lead to fewer learning rounds and equivalence queries than the state-of-the-art approaches?

RQ$_{1.4}$ Does variability-aware learning lead to fewer resets than the state-of-the-art approaches?

RQ$_2$ How can we classify previously unseen behaviour to multiple variants of a VIS?

RQ$_{2.1}$ How accurately can we identify process variants based on their traces?

RQ$_{2.2}$ What is the performance of LSTMs versus that of GRUs for process traces classification?

To address these research questions, we propose a two-phase framework. Figure 2 is used throughout this manuscript as a guide, and will be detailed step by step. Our contributions are divided into two parts: Variability-L^* and VaryMinions.

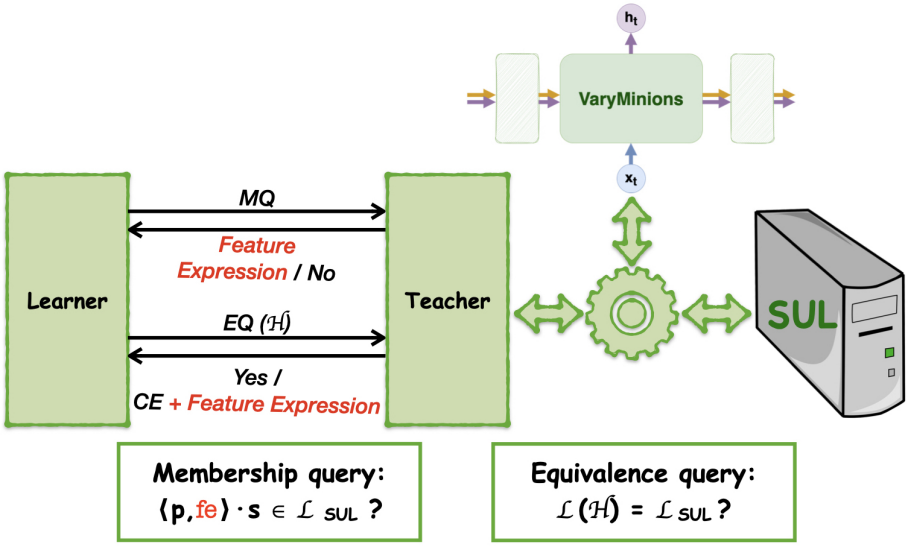

Figure 2: LiFTS General Framework

we make the following contributions:

Variability-L^* Contributions.

(i) Featured Deterministic Finite Automaton (FDFA), as a variation of Featured Transition Systems (FTSs), allowing finite behaviour and determinism by construction;

(ii) $Featured - L^*(FL^*)$, a new variation of the L^* algorithm, to automate the behavioural modelling of an SPL, considering variability from the early stage of learning;

(iii) LiFTS, an implementation of FL^*, which successfully learns 5 distinct case studies (with 5 to 24 input symbols) within a short time frame, ranging from a few seconds to less than two hours for each case study;
(iv) a first comparison with the original algorithm and the latest state-of-the-art approaches, demonstrating a significant reduction in the number of queries when variability is handled explicitly;
(v) a procedure to ease FTS and FDFA visualisation, by leveraging the capabilities of a Neo4J graph database.

VaryMinions Contributions.

(i) the first family-based approach, which we called VaryMinions, to map execution traces to variants of a system. We showed empirically that VaryMinions can distinguish 50 variants from 5,000+ event traces per variant;
(ii) a detailed account on the usage of LSTMs and GRUs on six different datasets, describing business processes and course management system variants, showing that we can identify the variant(s) producing an event trace with high accuracy ($> 80\%$);
(iii) four datasets openly available and based on Claroline [61, 64, 65] and containing $2 * 10$ and $2 * 50$ configurations with 5,000 traces per configuration;
(iv) a characterisation of the learning difficulty based on the behaviour shared amongst event traces.

Open Science Policy. We also provide replication packages [83, 87] with an implementation of FL^* and VaryMinions, as well as presenting all our datasets and results of our experiments.

Thesis Outline

This thesis is divided in five parts:

Part I provides the background of this thesis. Chapter 1 introduces VISs and how we can model them. Chapter 2 focuses on various techniques of model inference. Then, in Chapter 3, we present some machine learning techniques and establish their connections to VIS engineering.

Part II offers an adaptation of the L^* algorithm, to learn variability-aware behavioural models (answering $\mathbf{RQ_1}$). We present an overview of our solution in Chapter 4. Chapter 5 formalises essentials concepts required by Chapter 6 to present the new algorithm we propose. Chapter 7 presents different case studies. Chapter 8 is dedicated to the implementation of the algorithm and its evaluation on the case studies. We discuss amelioration perspective in the algorithm and threats to validity in Chapter 9.

Part III leverages ML techniques to map variability and behaviour (answering $\mathbf{RQ_2}$). Chapter 10 motivates VaryMinions and gives an overview of its architecture. Chapter 11 describes the implementation of VaryMinions, which

is evaluated on several datasets in Chapter 12. Chapter 13 discusses some threats to validity and possibilities for future work.

Part IV concludes this thesis and presents future research perspectives in Chapter 14.

Publications

The content of this thesis is based upon, reuses and extends the following peer-reviewed publications of the author:

Conferences

[85] Sophie Fortz, Fred Mesnard, Etienne Payet, Gilles Perrouin, Wim Vanhoof, and Germán Vidal. An SMT-Based Concolic Testing Tool for Logic Programs. In Keisuke Nakano and Konstantinos Sagonas, editors, **15th International Symposium on Functional and Logic Programming (FLOPS 2020), Akita, Japan**, pages 215–219, Cham, sep 2020. Springer International Publishing

[148] Edilton Lima dos Santos, Sophie Fortz, Pierre-Yves Schobbens, and Gilles Perrouin. Behavioral Maps: Identifying Architectural Smells in Self-adaptive Systems at Runtime. In Patrizia Scandurra, Matthias Galster, Raffaela Mirandola, and Danny Weyns, editors, **Software Architecture**, pages 159–180, Cham, 2022. Springer International Publishing

Workshops

[72] Edilton Lima dos Santos, Sophie Fortz, Gilles Perrouin, and Pierre-Yves Schobbens. A vision to identify architectural smells in self-adaptive systems using behavioral maps. In Robert Heinrich, Raffaela Mirandola, and Danny Weyns, editors, **4th Context-aware, Autonomous and Smart Architectures International Workshop (CASA 2021)**, 15th European Conference on Software Architecture (ECSA 2021), Växjö, Sweden, September 2021. CEUR Workshop Proceedings

[86] Sophie Fortz, Paul Temple, Xavier Devroey, Patrick Heymans, and Gilles Perrouin. VaryMinions: leveraging RNNs to identify variants in event logs. In Apostolos Ampatzoglou, Daniel Feitosa, Gemma Catolino, and Valentina Lenarduzzi, editors, **Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution, Athens, Greece, 23 August 2021**, pages 13–18. ACM, 2021

[82] Sophie Fortz. LIFTS: Learning Featured Transition Systems. In **Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume B, Leicester, United Kindom**, SPLC '21, page 1–6, New York, NY, USA, 2021. Association for Computing Machinery (Doctoral Symposium)

[73] Edilton Lima dos Santos, Sophie Fortz, Pierre-Yves Schobbens, and Gilles Perrouin. Identifying Architectural Smells in Self-Adaptive Systems at Run-

time. In **13ème édition de la Conférence francophone sur les Architectures Logicielles (CAL)**, Vannes, France, June 2022

[84] Sophie Fortz. Variability-aware Behavioural Learning. In **Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume B, Tokyo, Japan**, SPLC '23, New York, NY, USA, 2023. Association for Computing Machinery (Doctoral Symposium)

Part I

Background

CHAPTER 1

MODELLING VARIABILITY-INTENSIVE SYSTEMS

Over the years, systems with a single purpose tend to disappear in favour of configurable systems. Many software applications are designed to answer a common objective but within a **different socio-technical environment**. Required functionalities, hardware configurations, running operating systems, local legislation, and availability of external components, are some of the common sources of **software variability**. **Variability-Intensive Systems (VISs)** are software-based systems whose purpose is to address this diversity of customer needs and usage contexts. VISs have the ability to be customised according to specific needs, through the (de)activation of different options. We call these options **features**. The activation of **features** change the structure of the system and can affect its behaviour by adding, modifying or suppressing some functionalities [237]. An assignment of all features create a software **variant**. Features can be assigned to different types of values, for example integers denotes a feature in a certain quantity. In this thesis, we focus on Boolean assignment, suggesting that a feature can either be activated or deactivated.

Addressing variability proactively during software engineering (SE) activities means shifting from reasoning on a single system to reasoning on a **family** of systems. Using an appropriate variability approach can yield important economies of scale and quality improvements [177].

Conversely, variability can also be a curse, especially for **Quality Assurance (QA)**, *i.e.,* verification and testing of such systems. VIS QA activities must ensure that each variant behaves correctly, which is challenging since their number grows exponentially with the number of features. In practice, companies have limited budget. For example, in a study of JHipster (a Web development stack) [107], developers could only test 12 configurations due to budget limitations. Verifying or testing each variant is therefore impossible in most practical cases [107]. The VIS community tackled this combinatorial explosion issue by providing efficient **model-based** QA

techniques. Model-driven Engineering (MDE) [132] is a powerful SE paradigm advocating the use of **models** to face software complexity by abstraction and separation of concerns. The more formal the models, the more automation possibilities. For VIS modelling, we rely on compact **structural models** that describe valid combinations of features (*i.e.,* with respect to the system constraints) and **behavioural models** that describe shared behaviours among variants.

The following section introduces different application domains of software variability. Then, we present the structural and behavioural modelling of VIS.

1.1 Variability-Intensive Systems

Variability can be defined in terms of space (*e.g.,* different components with a common purpose but different implementations) and time (*e.g.,* different versions of the same component) altogether. **Variability-intensive systems** include, for example, operating systems kernels [162, 190], code generators [30, 206], or web-based frameworks [107, 183]. In this thesis, we will focus on three application domains of VIS: software product lines, configurable processes and adaptive systems.

1.1.1 Software Product Lines

To facilitate the design, development and QA of VISs systems, we can use structured approaches like **Software Product Lines (SPLs)** [10, 177]. The concept of SPLs draws inspiration from mass customisation engineering combined and a shared development platform. This combination not only enables the efficient reuse of technology and facilitates large-scale production, but also empowers the customisation of products to meet the specific needs of individual customers. SPLs consider a global base of software artefacts for a family of software systems and allow deriving **variants** through the (de)activation of **features**. SPL engineering separate two concerns: domain engineering (Definition 1) and application engineering (Definition 2). The main artefact produced by the domain engineering is the feature model of the SPL. The FM is then exploited to derive products in the application engineering phase. Figure 1.1 gives an overview of the complete SPL framework, as defined by Pohl *et al.* [177]. Apel *et al.* [10] introduce another representation (Figure 1.2), where they distinguish between the problem space, which involves analysing the domain and determining requirements, and the solution space, which encompasses the generated artefacts, such as models and derived products.

Definition 1 (Domain Engineering)**.** Domain engineering is *"the process of software product line engineering in which the commonality and the variability of the product line are defined and realised"* [177].

Definition 2 (Application Engineering)**.** Application engineering is *"the process of software product line engineering in which the applications of the product line are built by reusing domain artefacts and exploiting the product line variability"* [177].

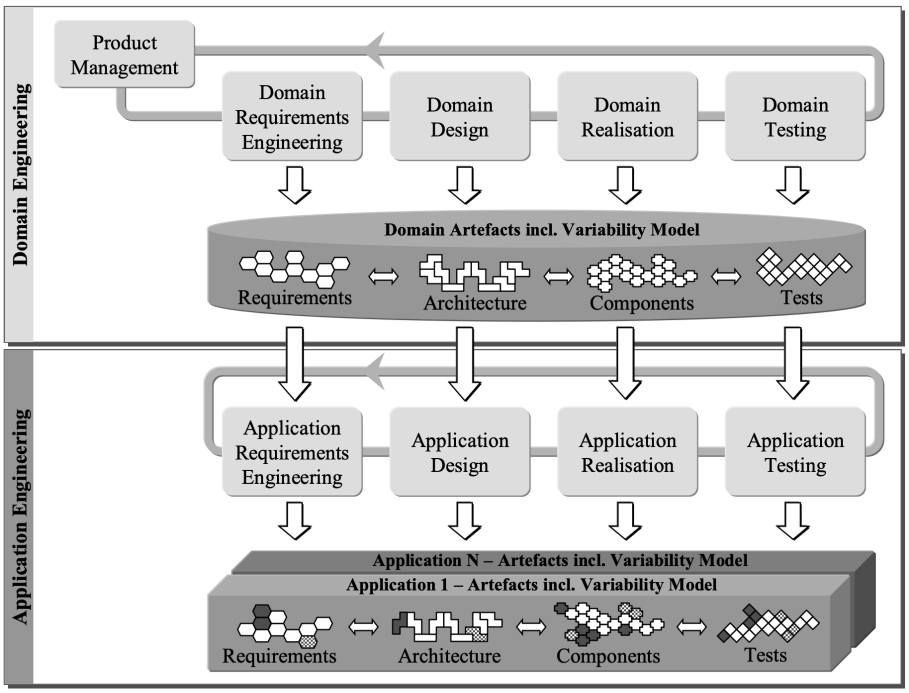

Figure 1.1: Software product lines engineering framework [177]

Software product line engineering has many benefits such as reduced development costs, reduced time to market and reduced maintenance effort, since error corrections are propagated throughout the whole family. Customers have access to customised products for a lower price. Moreover, new products have similar look and feel, which can help overcome resistance to change. Figure 1.3 highlights that the extra effort required for SPL development pays off quickly with the growing number of products.

However, validating these systems is generally difficult because enumerating all variants, whose number **grows exponentially** with the number of options, is generally infeasible. Validation is costly and for most SPLs, practitioners need to carefully select what they want to test. For example, JHipster is an open-source generator for Web applications with 48 features and more than 26,000 variants. However, the development team of JHipster only have budget to test 12 products [107]. This problem amplifies in context of continuous development, when product releases are closed.

Being able to locate variations is also an essential part of any re-engineering endeavour [15] and is naturally useful for testing techniques, notably to sample which variants should be tested [107].

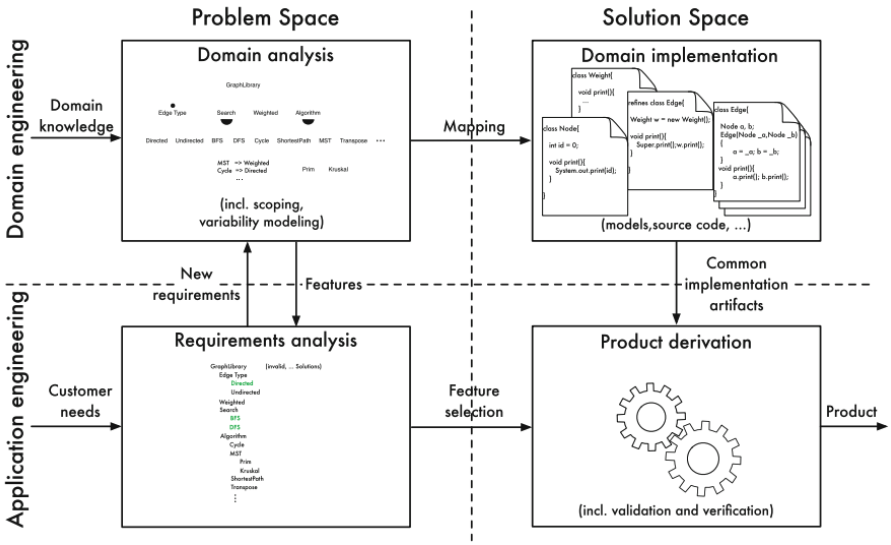

Figure 1.2: An engineering process for software product lines [10]

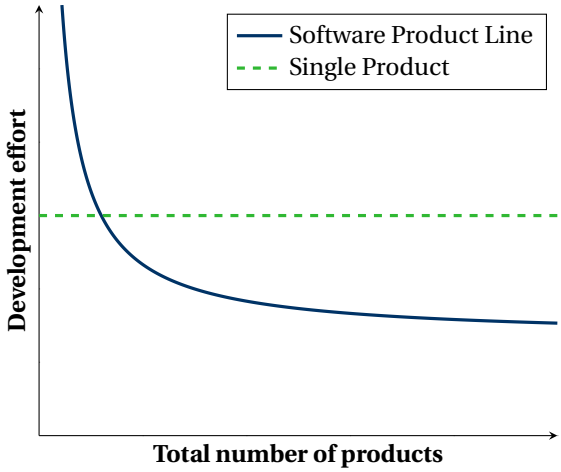

Figure 1.3: Single product development effort, with and without SPL engineering

1.1.2 Configurable Processes

Business processes [219, 220] capture the activities of every profit or charitable, public or private organisation, coordinating humans and software to collectively deliver value. As organisations evolve, new needs appear depending on environmental and human factors. For instance, a university needs to handle a change in the law about reimbursing travel expenses or an insurance company must adapt to cover electric vehicles. Therefore, organisations must adapt their processes and sometimes, act with several processes in parallel (*e.g.*, changes are progressive and not applied everywhere at the same time). These needs lead to the emergence of

process variants or **configurations**, differing in their control flow or performance while having commonalities with the original processes. Analysing the specificities and commonalities of process variants allows scale economies and helps practitioners to define new custom variants or maintain existing ones [204]. It can also help to improve the general business process. For example, this analysis can lead to a better management of shared resources. Similar process variants gather into **Process Lines** or process families and can be modelled using different formalisms [181]. In this study, they catalogued 23 variability mechanisms for configurable process modelling. They proposed a taxonomy composed of four groups:

- **Node configuration:** the behaviour of nodes tagged as "configurable" can be restricted at customisation-time (*e.g.*, removing activity or turning an "OR" gateway into a "XOR" gateway);
- **Element annotation:** annotations (*i.e.*, Boolean properties over a feature model) can be attached to elements of the configurable process model;
- **Activity specialisation:** a conceptual process model contains abstract activities which can be specialised to restrict the process behaviour;
- **Fragment customisation:** change operations (groups into sequences) can be applied to the configurable process model to add, delete or modify the process.

Node configuration, element annotation and activity specialisation are restrictive approaches, in the sense that they define a general (abstract) model whose behaviour is restricted. Fragment customisation allows both behaviour restrictions and behaviour extensions.

1.1.3 Adaptive Systems

To adapt to evolutions occurring **at runtime**, *e.g.*, in user needs or in evolving resource constraints, **Self-adaptive systems (SASs)** change their behaviour and structure dynamically according to reconfiguration plans and goals. For example, SASs are often used in internet-of-things (IoT) and Smart Homes environments to face problems of limited connectivity, hardware heterogeneity, changes of user preferences, etc. **Dynamic Software Product Line (DSPL)** [25,36] engineering implements SAS by combining different architectural fragments via feature binding/unbinding at runtime [192]. DSPLs are challenging to validate because the number of possible configurations grows exponentially with the number of features, and this problem is worse if the DSPL can self-update (*e.g.*, , by downloading new features to interface with a sensor newly plugged into the system) [37]. Particular runtime conditions may provoke **unwanted interactions** amongst features at the behavioural level [17, 49, 196] and the **combinatorial explosion** of variants (*w.r.t.* the number of features) makes these unwelcomed interactions difficult to detect. While the feature interaction problem is well-studied for systems where features are bound at specification or design time [11, 17, 18, 47, 49, 108, 157], runtime interactions are less explored [37, 175]. The (re)configuration process may also negatively impact the system's architectural qualities, exhibiting architectural bad smells (ABS), implying a

reduction of the system maintainability [57,150]. Moreover, some smells may appear only in particular runtime conditions. Since SAS do not document these features in their source code, design time smell detection ignores them and risks reporting smells that are different than those observed at runtime.

1.2 Structural Models

Various approaches exist to model the structural variability of VISs. A very popular one, defined in the SPLs community, is **Feature Models** (FM). FMs were introduced 30+ years ago [128] and later formalised by the SPL community (*e.g.,* [45,53,159,185, 186]). They equipped FMs with automated analyses [24] and comprehensive tool support [178]. Over time, more sophisticated FM dialects were proposed [22,75].

An FM declares the features of a VIS at a very abstract level and constrains how they can be combined to form valid system variants. As such, it does not aim at modelling the behaviour of a VIS, only the authorised combinations of features, *i.e.,* **structural variability**. Visually, FMs are tree-like diagrams where nodes represents features and edges are cross-tree constraints. Features are decomposed hierarchically: leaf of the tree are concrete features and upper nodes are usually abstractions. Constraints are defined using typical Boolean operators ("AND", "OR" and "XOR"). Each branch of the tree can be marked as mandatory or optional.

Example 1.2.1 (Feature Model of a Coffee Vending Machine)**.** Suppose we are interested in providing to various institutions customised coffee vending machines, catering to the unique preferences of their employees. Each institution has its own beverage options, which can include tea, coffee, cappuccino, or a combination of these. The vending machines should support payment in either euros or dollars (but not both). Optionally, the machines can be equipped with a ring that signals when the drink is ready. Figure 1.4 presents the FM of this simple configurable beverage vending machine. Nodes "VendingMachine", "Beverages" and "Currency" are abstract features (used for decomposition purpose), while the remaining nodes represent concrete features.

As the number of possible variants increases exponentially with the number of options available, such compact representation of the variability of a VIS enables various kinds of analysis, including counting the number of possible variants, detecting that can never be selected (*a.k.a.* dead options), *etc.* For instance, the coffee vending machine of Figure 1.4 counts already 28 possible (distinct) configurations, yet it only contains 6 features. This number is very small compared to real-world VISs. For example, Claroline (the course management system used in the University of Namur) has more than 5 million possible configurations for 44 options.

More than a decade of effort has been devoted to extracting options from VIS artefacts. Various studies have attempted to learn structural variability, *e.g.,* feature models. We can highlight different methods, such as natural language analysis [145] or static analysis of variant configurators [191]. Other studies learned FM from variant catalogues (product tables), either using data mining [4,5] or evolutionary algorithms [152]. These techniques were summarised in a recent systematic literature

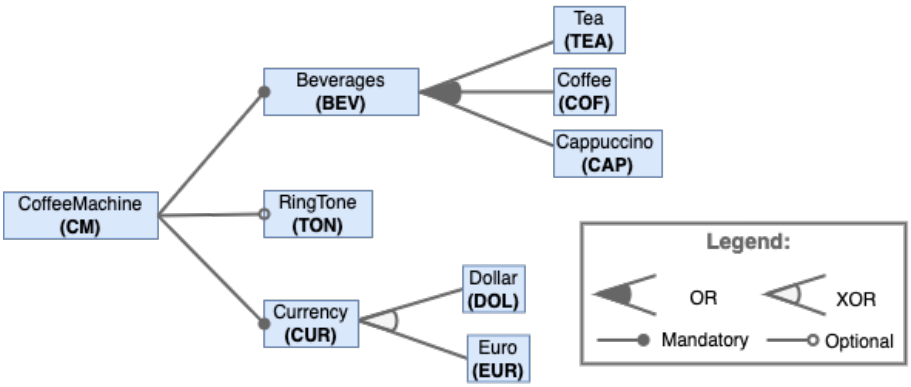

Figure 1.4: Feature Model of a coffee vending machine [44, 55, 78]

review [155]. Besides, Ramos-Gutiérrez *et al.* [180] use process mining to retrieve the process of configuring an SPL.

There also exist model-based approaches to recover an architectural model of a VIS (*e.g.,* [14, 133, 147]). This can be useful when the system was not designed using the SPL development approach from the onset (but *e.g.,* by using a clone-and-own approach) and when we want to perform complex maintenance or evolution tasks.

1.3 Behavioural Models

When it comes to model-based testing and formal verification of VISs, modelling structure is insufficient. The behaviour of the system must also be accurately captured. One of the key challenges for behavioural modelling of VISs is to avoid modelling each variant separately, as this can lead to a combinatorial explosion caused by feature combinations. However, by adopting a family-based reasoning approach, we can overcome this challenge. This approach allows us to test each behaviour only once, regardless of the number of variants that exhibit the same behaviour, while still ensuring efficient testing coverage. Moreover, in the event of bugs, any necessary corrections can be directly propagated to all relevant products, streamlining the debugging process. To this end, several state-based languages were proposed to model all the variants of a system into a single behavioural model. The powerful combination of feature models and state-based representations supported efficient and scalable QA tasks for VIS, including model-checking and model-based testing.

1.3.1 Modal Transition Systems

At the origin, Larsen *et al.* [141] introduced **Modal Transition Systems (MTSs)** as an extension of process algebra to handle nondeterministic and concurrent processes. They enable the expression of loose or partial specifications within a process algebraic framework. They offer an operational interpretation that restricts the

transitions of possible implementations, distinguishing between necessary and admissible transitions. MTSs provide a powerful mean to express and analyse the behaviour of complex systems with nondeterminism and concurrency. In 2006, Fischbein *et al.* [81] proposed to model the behaviour of an SPL with an MTS. For the first time, the behaviour of a complete family of products is expressed within a single unified representation.

An MTS is a Labelled Transition System [131] where each transition can be labelled either with *"may"* or *"must"*. A behaviour shared by all configurations is then **mandatory** and is represented with *"must"* transition. A *"may"* transition depicts a variation point of the system: the transition can **optionally** be taken (or not). Figure 1.5 shows an example of MTS, depicting a family of coffee machines (based on the FM example depicted in Figure 1.4). Mandatory and optional transitions are shown with plain and dotted arrows respectively. MTSs have been used to model and verify behavioural properties over the products of a SPL [13, 70, 213], which cannot be addressed by a (static) feature model.

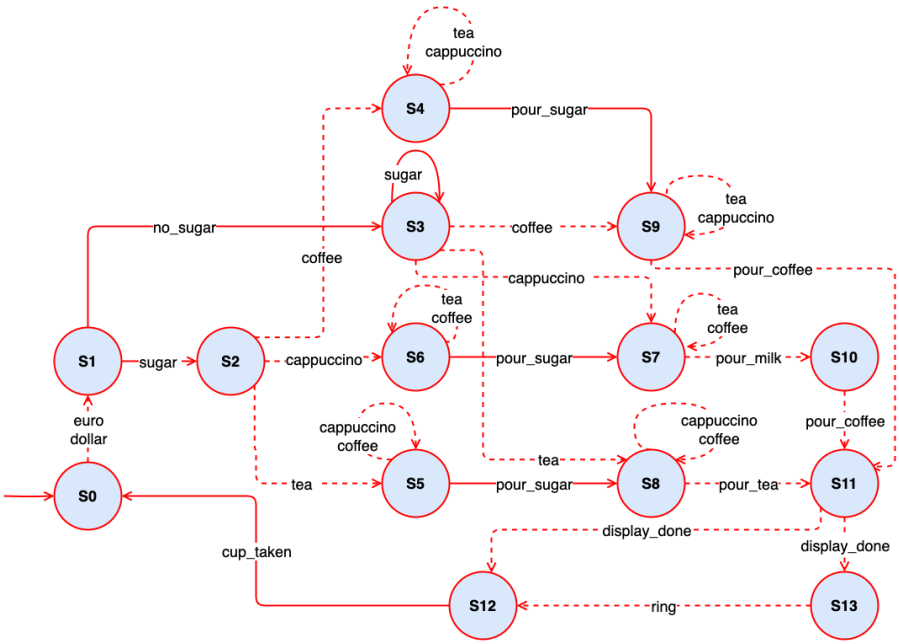

Figure 1.5: Modal Transition System of a coffee machine family [44, 78, 213]

1.3.2 Featured Transition Systems

About a decade ago, **Featured Transition Systems (FTS)** [48, 51] were introduced as a formalism to represent, and reason on, the behaviour of SPLs. An FTS is a transition system [131] where each transition is labelled using annotations called **feature expressions**. Feature expressions are logical formulae referring to structural variability of the FTS, *a.k.a.* its feature model. They indicate which valid configurations of the

VIS can execute each transition. For instance, Figure 1.6 presents the FTS of the coffee vending machine described in the previous section (whose FM is presented in Figure 1.4). As can be seen from the feature expressions, only specific configurations can execute some transitions: *e.g.,* only vending machines with the ringtone option enabled (`TON`) can execute the `ring` transition from state 13 to state 12. FTS have been shown to significantly improve the possibilities and execution time of automated QA activities such as model-checking and model-based testing [46, 50, 68, 212]. Thanks to the refinement degree of feature expressions, FTSs are more expressive than MTSs [26, 226]. The transformation from FTS to MTS is thus only possible if additional constraints are added [210, 211]. By adding these constraints however, MTSs became equally expressive as FTSs. Another approach consist in transforming an FTSs into a set of MTSs [226].

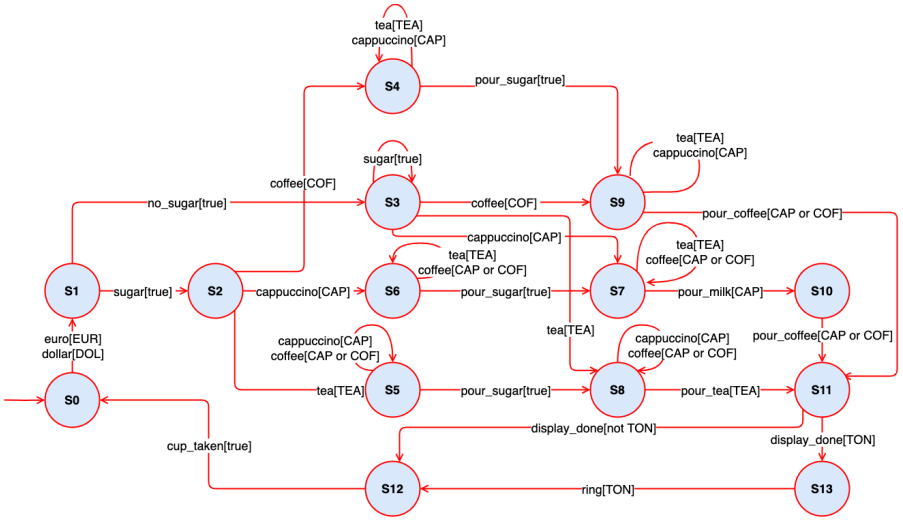

Figure 1.6: Featured Transition System of a coffee vending machine [44]

1.3.3 Featured Finite State Machines

A Featured Finite State Machine (FFSM) [88] is an extension of a finite state machine [98] for a complete SPL. As FTSs, FFSMs add feature constraints to specify subsets of products. Figure 1.7 depicts an example of an FFSM for a variant of the beverage vending machine, whose FM is shown by Figure 1.4. While we can compare MTS and FTS in terms of expressiveness, it is not possible to compare FTS and FFSM [88]. FFSMs have been used for model-based testing and model checking in several studies [88, 90]. Fragal *et al.* [89] introduced a significant improvement in the modelling of large SPLs by incorporating the concept of hierarchy. This approach greatly simplifies the modelling process, as each component can be represented as a separate FFSMs, which then contributes to the overall structure of the general FFSM. This hierarchical organisation enhances the management and comprehension of complex SPLs, leading to more efficient modelling and analysis.

In comparison, modelling large SPL with MTS or FTS is more challenging, due to their lack of hierarchy.

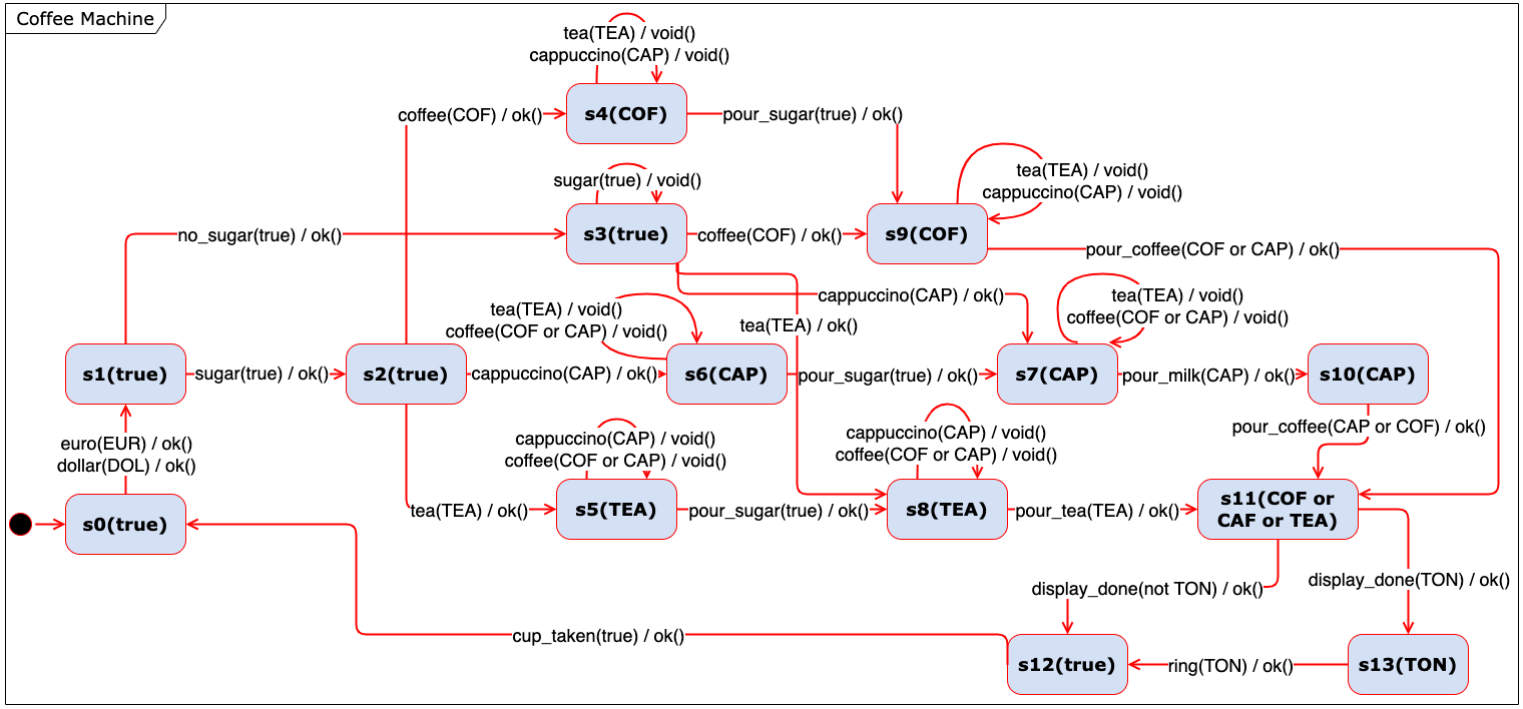

Figure 1.7: Featured Finite-State Machine of a beverage vending machine [55]

CHAPTER 2

BEHAVIOURAL MODEL INFERENCE

Model-based approaches for VIS quality assurance activities are only as good as the models they rely on [20]. Moreover, they require a significant amount of human effort and expertise to provide models in the first place. This effort may be unbearable for large legacy VIS and may yield outdated models if these systems continuously evolve. While this issue is not specific to VIS, it is further aggravated by the fact that we need to recover the models for the whole family of systems.

Behavioural modelling has proven valuable in the context of model checking and testing VISs (see Section 1.3). Several semantics for behavioural modelling of VISs already exist, highlighting the significance of this subject within the research community. Yet, until very recently, researchers primarily focused on learning structural aspects (Section 1.2). As a result, we have identified a gap in the literature regarding the learning of behavioural models for VISs. In this chapter, we present state-of-the-art techniques to automatically learn a behavioural model for single systems (*i.e.,* without variability). Subsequently, we delve into the few recent approaches that address the specific challenges associated with learning behavioural models for VISs.

2.1 Single Model Inference

According to Vaandrager [216], the problem of inferring behaviour from software artifacts is an active research area since 1956 and Moore's work. Model learning techniques fall into two categories: **white-box** approaches exploit source code while **black-box** approaches do not and rather exploit execution traces, which can either be obtained via **active** interaction with the systems or mined **passively** from logs.

2.1.1 Black-Box Approaches

Black-box approaches regard the system as an entity whose internal details remain unknown, making observation of its execution the sole means of harnessing its potential. For instance, by providing inputs and monitoring the corresponding responses (assuming observability in the form of a response or a set of logs), these real-time observations can be leveraged through **active learning** and interactions. This enables real-time control or guidance of inputs, with the objective of diversifying system behaviour. Alternatively, a **passive approach** relies on pre-generated observations, where requesting new observations on-the-fly is not feasible.

Active Learning

Black-box approaches have been significantly influenced by the seminal L^* algorithm, which was introduced by Dana Angluin in 1987 [9]. This algorithm is specifically designed to actively learn a model that represents an unknown regular language. The resulting model can be represented by various types of data structures dedicated to language recognition, including finite state machines (FSMs) or automata. The L^* algorithm abstracts the learning process using a simple metaphor that defines two main components: the **Learner** and the **Teacher**. The Learner's objective is to progressively construct an automaton that accurately represents the **system under learning** (SUL). The Teacher acts as a middleware between the Learner and the SUL, facilitating communication between them. The Learner and the Teacher exchange information through queries, as illustrated in Figure 2.1.

There are two distinct types of queries serving different purposes. **Membership queries** are used to determine if a word (*i.e.,* a sequence of actions, belonging to the system's alphabet) belongs to the language being learned. Based on the observations, the algorithm can assess the soundness of the constructed hypothesis. If the hypothesis seems reasonable, the algorithm can then request an **Equivalence query** from the Teacher. This query enables the algorithm to validate its hypothesis against the actual system behaviour and ensure the correctness of the learned model. If the hypothesis and the SUL are not equivalent, the Teacher provides a counterexample (*i.e.,* a word that is in the language but not accepted by the hypothesis automaton, or vice versa). The Learner can then use this counterexample to explore a new portion of the system and repeat the whole process until equivalence is achieved. In practice, counterexamples can be obtained through conformance testing and monitoring.

Since the teacher is able to directly interact with the SUL, it requires no further access to the source code. The approach relies solely on test executions: input sequences are generated and the algorithm computes output sequences on-the-fly. Due to these two characteristics, L^* is classified both as a black box and an active approach.

Angluin's algorithm is a powerful theoretical framework that has given birth to numerous optimised versions and extensions. For example, the algorithm has been adapted for the learning of nondeterministic finite automata [28], mealy machine [144, 169, 189], register automata [1, 2, 38, 39], input-output transition systems [230] or probabilistic subsequential transducers [6]. The *LearnLib* frame-

work [158, 179], widely used in the research community, provides various Java implementations of L^* and its extensions. AALpy [163] is another framework for active automata learning, implemented in Python.

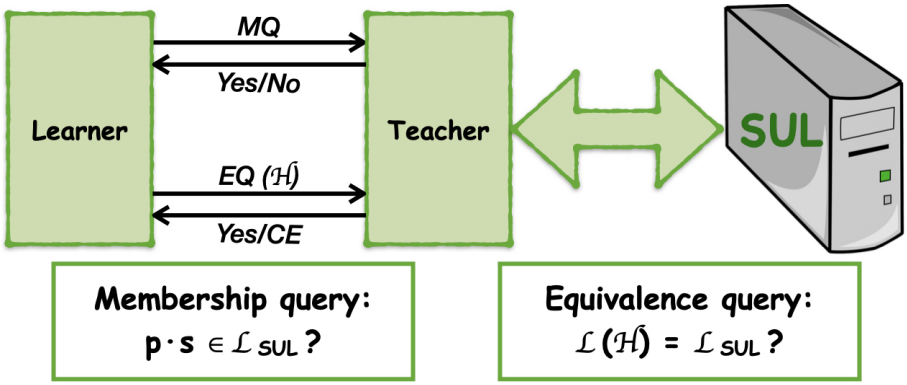

Figure 2.1: L^* algorithm

Passive Learning

In passive approaches, the learning component exclusively uses existing execution traces. In this case, the model is incomplete and can only contain the **observed behaviour** of the system [214].

Directly influenced by L^* algorithm, the *Regular Positive and Negative Inference (RPNI algorithm)* [74, 140, 172] operates within a passive setting, relying solely on pre-existing examples. Similar to L^*, RPNI consider both positive inputs (*i.e.,* words from the language being learned) and negative inputs (*i.e.,* words not belonging to the language). In fact, Gold [100] demonstrated that positive examples alone are insufficient for learning regular languages.

Evidence-driven state merging algorithm [116, 138] is another approach that has gained interest, by winning several automata learning competitions (*e.g.,* Abbadingo [139] and STAMINA [232]). State merging algorithms work by reducing execution trees into smaller graphs, using merging and mappings. Verwer *et al.* provide a tool, *Flexfringe* [229], implementing this idea. There also exists statistical methods based on N-Grams [215] and usage-based model [115, 198] (*i.e.,* model representing the usage of a system as a Discrete-Time Markov Chain). Process discovery algorithms studied by the process mining community [143, 218] also fall into the category of passive approaches.

2.1.2 White-Box Approaches

White-box approaches rely on program analysis. They allow learning enriched automata by retrieving more precise and complete information (*e.g.,* value of state variables). Shoham *et al.* use static analysis to mine Internet API specifications [193];

Fraser *et al.* also use static analysis to infer object usage and thereby generate more meaningful tests [91].

Black-box and white-box approaches are complementary and can be orchestrated in a grey-box fashion. For example, Howar *et al.* use a mix of static, dynamic and **concolic analysis** (a mix of symbolic and concrete execution) to learn safe interfaces for critical embedded systems [120]. In a recent article, Howar *et al.* suggest a gray-box scenario where predicates or guards are exploited to guide black-box learning [121].

2.2 Variability-Aware Behavioural Inference

In contrast to FM learning, learning behavioural models of VIS (*i.e.,* at the family level) is still in its infancy. In different contexts, related approaches are that of Buijs *et al.* [34] which use genetic algorithms to combine process models mined from event logs of (a few) different variants, and that of Greenyer *et al.* [103, 104] to check and synthesise controllers for VIS from scenarios (message sequence charts). In the literature, only two distinct contributions really attempt to learn the behaviour of an entire SPL.

Usage Models

In 2015, Devroey *et al.* [63, 65] designed the first approach to retrieve a behavioural model of an SPL. They aimed to perform statistical prioritisation of FTS-based tests. Their technique, based on usage models (Markov chains) inferred from logs, learns the structure of an FTS. Then, they manually annotate the FTS with **feature expressions**, requiring **significant human effort and expertise**. Despite their use of Markov chains to automate the learning of the FTS structure, these annotations still require a lot of time for reasonable-size systems.

Partial-Dynamic L^*_M and $FFSM_{Diff}$

Damasceno *et al.* [54] proposed an approach to keep VIS models up-to-date. They defined an adaptation of the classical learning algorithm $L*$ [9], called Partial-Dynamic L^*_M (∂L^*_M). In ∂L^*_M, the algorithm is instrumented to merge individual models of each variant into a single family model (*i.e.,* an FFSM [106]).

Based on this idea, they learned featured finite state-machine models **from individual models** [56]. While $FFSM_{Diff}$ is fully automated, their approach requires learning each product model separately with a modified version of Angluin's algorithm, called ∂L^*_M. This approach requires to merge the individual models of each product. Since the number of products is exponential over the number of features, $FFSM_{Diff}$ is therefore limited to a few sampled variants.

More recently, Tavassoli *et al.* [201] improved the algorithm by reusing not only the previous model but also internal data structures (that we will describe in Chapter 5). They thus take variability into account at a previous stage, but they still need to consider each variant separately.

In Section 6.5, we will compare classical L^*, the approach by Tavassoli, Damasceno *et al.* [54, 56, 201], and our approach.

CHAPTER 3

MACHINE LEARNING AND VARIABILITY

Machine Learning (ML) algorithms are statistical approaches that leverage patterns and relationships in previous data to make predictions or classifications. These algorithms learn from a training dataset, which consists of input data along with their corresponding labels or outcomes. By analysing the patterns in the training data, ML algorithms infer a model that can generalise and make predictions on new, unseen data. In a classification task, for example, an ML algorithm can learn to categorise new data points into different predefined categories based on the similarities it has identified from the training data. The algorithm looks for common features [1] or characteristics shared by the instances within each category and uses them as indicators to classify new instances. Different families of machine learning algorithms exist, such as decision trees, random forests, support vector machines, linear regressors, *etc.*

Raw training data usually contains noise and non-relevant information for accurate prediction. To address this, it is common to extract a new, compact representation of the data before applying ML approaches. This process involves identifying and selecting predefined features (or characteristics). This feature extraction step plays a crucial role in enhancing the performance and predictive capabilities of ML algorithms. Historically, ML features are defined by domain experts.

Example 3.0.1 (Iris dataset)**.** The Iris dataset [2] contains 150 instances of iris plants. Domain experts have classified the iris plants into three distinct classes based on

[1] Here, we are discussing the concept of an **ML feature**, which refers to a property that characterises an entity, rather than a **VIS feature**, which pertains to the functionality of a software system. It is important to differentiate between these two types of features in order to avoid confusion and ensure clarity in our discussion. The distinction is further elaborated upon by Temple *et al.* [208], where they delve deeper into this subject, providing valuable insights into the significance and implications of such differentiation.

[2] https://archive.ics.uci.edu/ml/datasets/iris

distinguishing attributes, including the length and width of their sepals and petals. Other characteristics of the iris plant, such as its colour, were intentionally ignored as they are not relevant for determining the iris species. Therefore, the input dataset for iris plant classification is structured as tabular data with 150 rows representing the instances and 4 columns representing the features. Each row corresponds to an iris plant, and the columns contain the measurements of sepal length, sepal width, petal length, and petal width, respectively. These four features provide the necessary information for training machine learning models to accurately classify the iris plants into their respective classes.

In contrast, a subset of ML techniques called Deep Learning (DL), can automatically infer complex features while training, but at the cost of more computational resources and time. Figure 3.1 depicts a Venn diagram illustrating the relationship between DL, ML and AI, along with a representative applications for each. Over the past decade, DL techniques have outperformed traditional ML algorithms in different tasks (*e.g.,* image processing, text processing, sound processing), but also demonstrated remarkable achievements in new applications (*e.g.,* assistance in driving autonomous vehicles, playing Go, automatic translation). Temple *et al.* [208] compared the explicit (traditional ML) and implicit (DL) feature management in the context of VISs. Thanks to their capability to model and handle complex relations, neural networks are at the centre of attention of DL techniques. Different neural network architectures exist, each tailored to specific tasks. For instance, convolutional neural networks (CNNs) excel at image processing while recurrent neural networks (RNNs) [182, 187] are better at handling sequential data (such as text or speech).

In the context of VISs, the number of possible variants grows exponentially with the number of options. Similarly, the number of traces a system can generate is supposed to be infinite. To reason on VISs artefacts (*e.g.,* products or execution traces), manual inspections are intractable. Instead, we can automate reasoning by relying on machine learning and deep learning techniques. In the remainder of this chapter, we introduce **Recurrent Neural Networks (RNN)** in general and related work of ML applied to software engineering.

3.1 Recurrent Neural Networks

An RNN is a sequence of multiple **units** (sometimes referred to as cells) which can convey data from one to another. Typically, RNNs start with an embedding layer that transforms input data into multi-dimensional vectors. However, when data sequences are too long, **vanilla** RNNs may face the so-called **vanishing or exploding gradient** problem [118]. This issue arises due to the nature of RNNs and their back-propagation mechanism during training. In the training process, prediction errors are propagated backward through the network from the output layer to the input layer. However, when sequences are too long, the errors tend to diminish exponentially as they move backward, causing the gradients to become extremely small, and as a result, the weights of the earlier layers may barely be updated or remain unchanged, leading to the vanishing gradient problem. Conversely, the gradient can

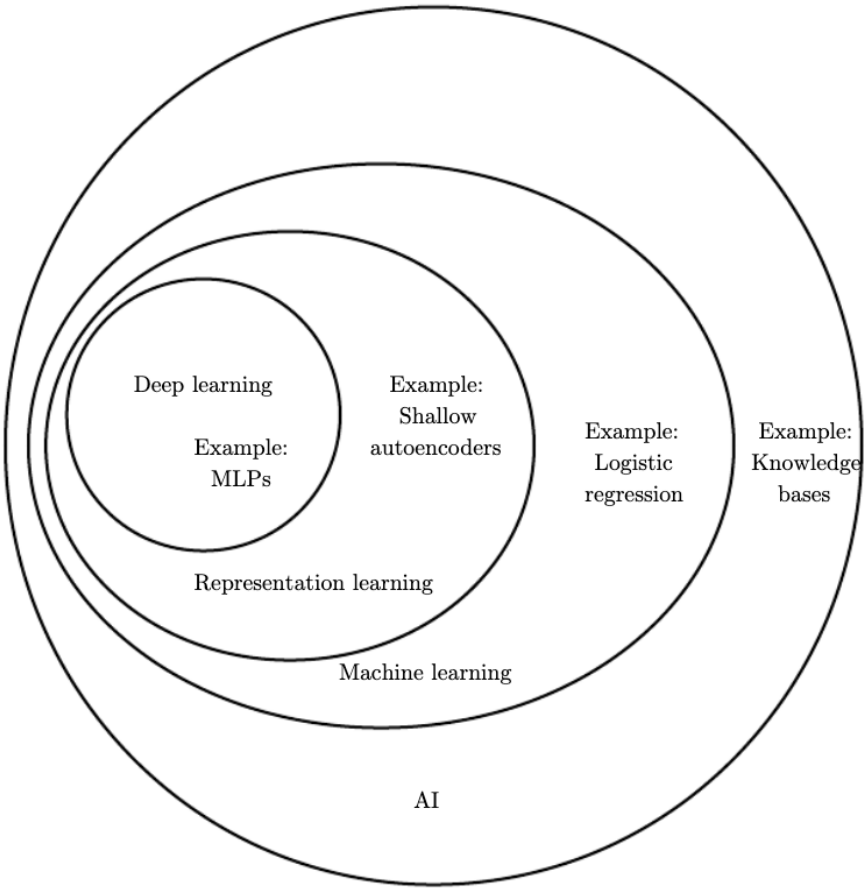

Figure 3.1: A Venn diagram illustrating the relationship between deep learning, representation learning, machine learning and artificial intelligence [102].

grow exponentially, yielding intractable computations, *a.k.a.* the exploding gradient problem. To address these issues and enable RNNs to handle longer sequences and long-term dependencies more effectively, two specialised RNN architectures have been introduced: LSTM [119] and GRU [41].

3.1.1 Long-Short Term Memory

LSTMs alleviate gradient issues [43, 118] by using gates to regulate the data flow and keep specific long-term data in memory. Figure 3.2 depicts an example of an LSTM unit. Inside one unit, gates regulate the data flow, deciding what data to keep and what to forget. Mathematically, gates are functions (*e.g.,* sigmoid or hyperbolic tangent) expressing the amount of data to keep. We can define several types of internal gates for different purposes. An LSTM unit (Figure 3.2) is composed of three different state variables and three different gates. The variables represent

respectively the input of the unit (*i.e.,* the matrix computed by the embedding layer, called x_t in the figure), the output (called h_t), and the unit state (called c_t). The latter acts as the long-term memory of the network, registering data from previous units to pass through to the next ones. Forget gates (on the left of the Figure) are used to convey data from the previous unit directly to the next one. In particular, it may set some values from the input (x_t) or from the memory (c_{t-1}) to 0, making the network forget this data. The input gate (in the middle) defines how much of the input data should be treated in the current unit. The final output of a unit travels through the output gate (on the right of the Figure). To avoid gradient explosion, LSTM units use a *tanh* function (above the output gate) to keep data in a small range of values (*i.e.,* between −1 and 1).

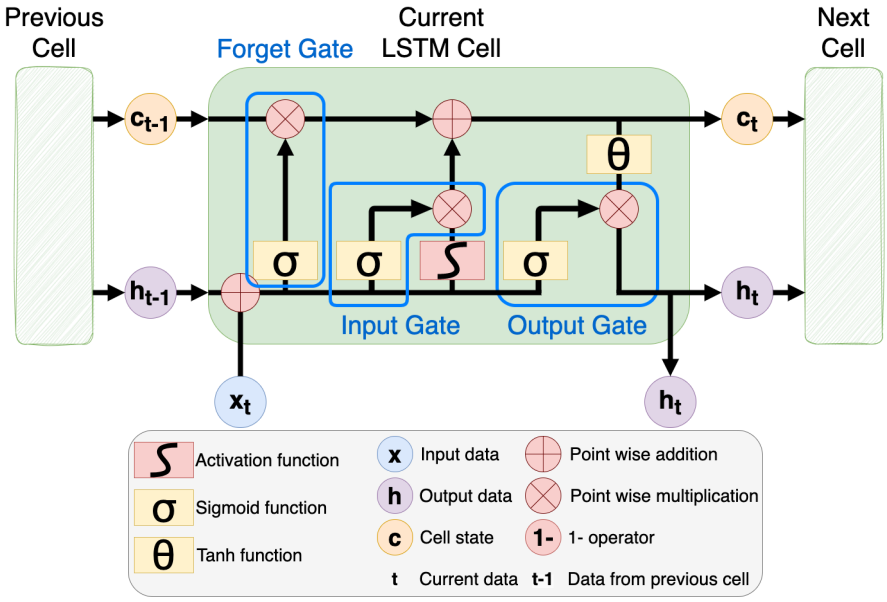

Figure 3.2: A Long-Short Term Memory Unit

3.1.2 Gated Recurrent Unit

Similarly to LSTMs, GRUs use gates to avoid gradient issues. In comparison with LSTM, GRU input and forget gates are merged together (Figure 3.3, on the right) and there is no output gate. Consequently, the output and unit state variables are also combined to a unique variable (named h_t in the Figure). GRU also offers a new type of gate (in the middle of the figure) expressing how relevant data from previous unit is for the current unit.

3.1.3 RNNs for software specifications

While RNNs are usually good fits to work on NLP tasks (*e.g.,* text classification [135, 151]), there is little work trying to use RNNs in the context of technical documents

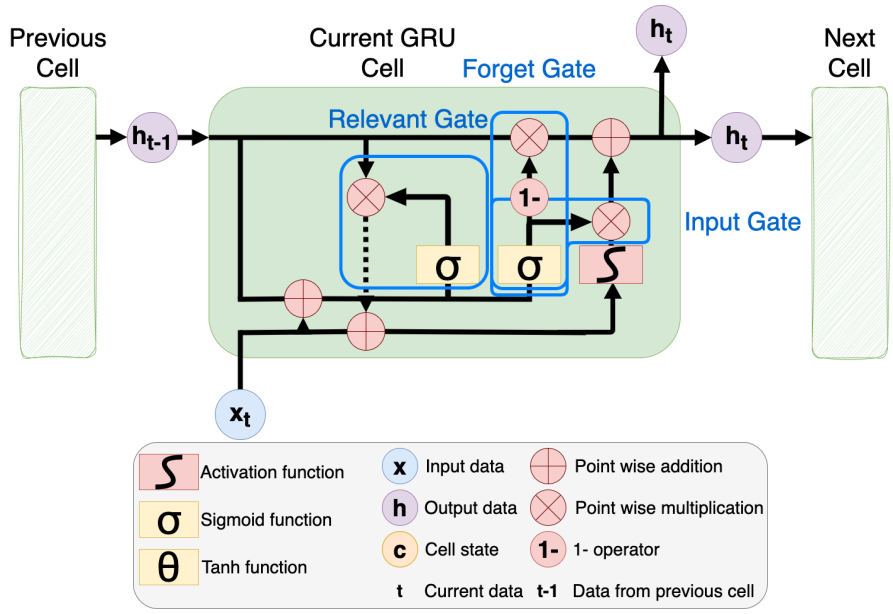

Figure 3.3: A Gated Recurrent Unit

or software specifications. Li *et al.* conducted a systematic literature review on extracting variants from text specifications [145].

To the best of our knowledge, none of the related works relied on RNNs but used other classification models (such as decision trees and association rules). Recently, Arganese *et al.* investigated ambiguity in natural requirements as variability points [12], with a mapping on words rather than complete sequences.

Nevertheless, we can compare technical documents to natural language. For example, execution traces are succession of events occurring in a specific order. In this context, an event does not appear randomly but depend on the previous succession of events. Sometimes directly from the few previous ones, sometimes because of an event that occurs way earlier in the trace. As such, we can consider them as text, *i.e.*, an ordered sequence of symbols that follows a given grammar. Thus, we can consider RNNs as an appropriate tool to treat them.

3.2 Machine Learning for VISs

ML techniques have been used in conjunction with business processes and SPLs for very different purposes. This section gives an overview of existing approaches where both ML and variable systems meet.

3.2.1 Machine Learning for Process Monitoring and Mining

Machine learning, in particular deep learning, has been notably used in business process monitoring. For instance, ML models can use past observations to predict

the next event in a process [69,156,203,205,228], the outcome of a process [31,76,136, 233], the remaining time [200,236], vulnerabilities and anomalies [29,112,168,170, 171] or even performance [173]. This vast research area, called **predictive business process monitoring**, attracted several literature reviews (*e.g.,* [111,167]). ML can also be used to optimise existing processes [79] or to get a compact representation of traces [32,33]. Recently, there has been interest in the interpretability of RNN models, specifically in a process mining context [110].

Han *et al.* [109] use LSTM to discover automatically business processes from textual documentation. However, their work is focused on single processes and does not highlight variability.

3.2.2 Engineering Configurable Processes

When trying to (reverse-)engineer configurable processes or even perform maintenance and/or evolution, some of the reported techniques rely on grammar-based or evolutionary algorithms, while others are machine learning (ML) oriented. The latter mostly consider tasks like clustering traces (*e.g.,* [197]). However, few techniques allow for retrieving a complete configurable process from event logs. Some approaches use genetic algorithms [34,137], but they are limited to a small number of variants. Another option is to use (configurable) process fragments to re-build the configurable model [16]. Sikal *et al.* propose a pattern for variability discovery during process mining, but this approach is only methodological at this stage [195].

In our case, we focus on the classification task. Bobek *et al.* [27] offer recommendations to configure variability-aware business processes at design time with Bayesian Networks. Clustering techniques have also been used [58,153,225] to perform classification tasks in an unsupervised way, *i.e.,* without knowing the classes to learn. Song *et al.* use dimensionality reduction techniques to improve trace clustering [197]. In our context, we want to specify the variants (*i.e.,* the classes) to learn. Finally, Hinkka *et al.* [117] aim at categorising traces into classes, using LSTMs and GRUs. However, their approach differs from ours on several points: (i) they define artificial classes, and (ii) they focus on binary classification.

3.2.3 Machine Learning for Variability-Intensive Systems

While there is a growing interest to employ ML techniques for VIS engineering [80, 174], to the best of our knowledge, classification of variants from behavioural traces using ML techniques has not been studied yet. ML approaches have been used to support performance prediction (*e.g.,* [7,19,105,127,194,217,238]), performance optimisation (*e.g.,* [71, 154, 227, 234, 235]), to improve the search for good and acceptable configurations (*e.g.,* [165,207,209]), and to predict unwanted feature interactions [134,146]. While some of these works target classification tasks, where configurations are the main focus, they do not take the behaviour of the studied systems into account. In such cases, configurations are treated as input parameters, and the primary objective is to find the most suitable configuration based on specific criteria, rather than understanding how structural variability impact the system's

behaviour. In contrast, in this thesis, we take a different approach by using ML techniques to thoroughly analyse the system's behaviour. Our objective is to gain a comprehensive understanding of the intricate relationships between configurations, feature interactions, and system behaviour. ML also supports usability prediction [231], attacks and vulnerabilities detection [3], and defect prediction [8, 199]. In particular, Strüder *et al.* demonstrated that artificial neural networks were suitable for this last task [199].

While ML can support VIS engineering, the converse, *i.e.,* applying variability-aware techniques to neural networks is also possible. For example, Ghofrani *et al.* [96, 97] proposed a new approach to reuse modules of deep neural networks without additional training. Along the same lines, Ghamizi *et al.* developed a framework to explore variability amongst different neural networks architectures and automated search-based techniques to find the optimal one for a given task [94, 95].

To conclude, we have discussed in Sections 1.2 and 2.2 various attempts to reverse engineer VIS from different artefacts, utilising ML techniques or other methods. These techniques operate at different levels, such as learning a variability model (Sections 1.2) and learning VIS behavioural models (Sections 2.2).

Another task in VIS reverse engineering is feature location, which involves mapping between options and VIS artifacts. Cruz et al. [52] categorise feature location techniques into three categories: static (based on source code), dynamic (based on execution traces), and textual (based on NLP). Some approaches even combine multiple techniques to enhance their accuracy and effectiveness, such as the hybrid approach proposed by Michelon et al. [160], which combines static analysis of source code with dynamic analysis of execution traces. It is important to note that feature location techniques differ in nature from the black-box approaches discussed earlier. While black-box approaches focus on understanding the system's behaviour without relying on internal details, feature location techniques are white-box approaches that primarily aim to map features with corresponding source code artifacts, often achieved through source code annotations. These techniques are commonly used for maintenance and evolution purposes, assisting developers in managing and modifying software systems. Notably, classical feature location techniques (*e.g.,* [52, 160]), do not use recurrent neural networks (RNNs).

Furthermore, model-based approaches have been proposed to recover the architectural model of a VIS [14, 133, 147]. These approaches are useful in scenarios where the system was not originally designed with the Software Product Line (SPL) paradigm in mind, but rather developed using a clone-and-own approach. In such cases, model-based techniques can facilitate complex maintenance and evolution tasks.

Part II

Variability L^*

CHAPTER 4

VARIABILITY L^* OVERVIEW

In Chapter 2, we have explored the existing literature on inferring behavioural models for individual systems and shown that several techniques have been developed in this domain. However, we identified a significant gap in the state-of-the-art when it comes to VISs. Existing techniques for VISs have not fully leveraged the power of variability mechanisms, leaving room for improvement and innovation in this area.

Given the access to a VIS running system and its feature model, the question we want to answer is thus **how can we accurately learn a model of its behaviour, without requiring learning separate variants or merging?** (**RQ_1**). To address it, we propose the first technique to learn the behaviour of a VIS which integrates variability at each stage of the learning process (See Figure 4.1).

More specifically, we divide this task in four research questions:

- **$RQ_{1.1}$ What amount of time does a variability-aware learning approach requires?** This question addresses the feasibility, efficiency and scalability of our approach.
- **$RQ_{1.2}$ Does variability-aware learning lead to fewer membership queries than the state-of-the-art approaches?** By limiting the number of queries, the learning process becomes more efficient and faster, thereby allowing better scaling on large systems.
- **$RQ_{1.3}$ Does variability-aware learning lead to fewer learning rounds and equivalence queries than the state-of-the-art approaches?** Equivalence queries are typically very expensive in a learning algorithm, requiring significant computational resources and time. Therefore, it is crucial to minimise their number whenever possible.
- **$RQ_{1.4}$ Does variability-aware learning lead to fewer resets than the state-of-the-art approaches?** Resets allow a system to come back to its original state,

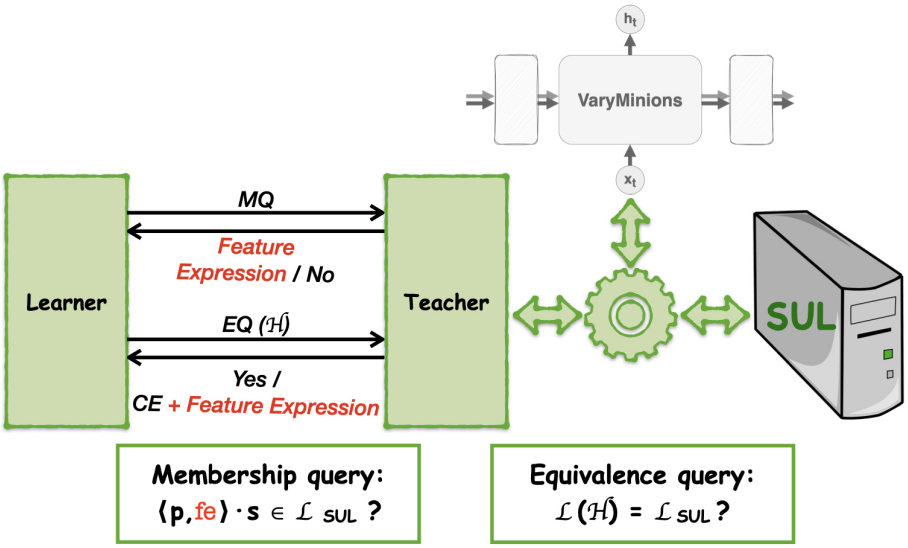

Figure 4.1: LiFTS overview: the **FL*** algorithm

as if we had just restarted it. Resets are difficult to avoid during the learning process, but they are time-consuming and should be avoided.

In this part, Chapter 5 explains L^* algorithm in details. Then in Chapter 6, we propose FL^*, a variant of L^*, that treats feature expressions as first-class citizen. We present different case studies in Chapter 7 and we evaluate our implementation of FL^* (LiFTS) on these in Chapter 8. Finally, we discuss potential further improvements (Chapter 9).

CHAPTER 5

GENERAL CONCEPTS AND NOTATIONS

In order to develop **RQ$_1$**, we need to formally define fundamental concepts, *a.k.a.* the building blocks of our contributions. To begin with, this chapter presents the feature model, featured transition systems and a variant of FTS, suitable for learning. Then, we describe Angluin's theoretical framework, such as principal definitions and data structures, and how we adapt them to cope with variability.

5.1 Feature Models

When considering each product configuration individually, verification and validation of VIS is an intractable approach. Some VIS like the Linux kernel having several thousands of features, and the number of possible variants grows exponentially with the number of features. Therefore, classical modelling approaches should be avoided, in the benefice of **family-based** approach. To model the structure of a product line, software engineers typically rely on feature modelling [128, 185]. A feature model (noted FM) captures the technical and business/application domain constraints specifying which combinations result in **valid** products (or configurations) [1]. These constraints are defined over (a set F of) features of the product line. Formally, an FM is a Directed Acyclic Graph. Nodes either represents concrete features, *i.e.,* they reflect an actual feature from F, or abstract features, *i.e.,* intermediate nodes used for decomposition purposes. Edges are split into two categories: "decomposition" edges relate a node bearing an operator (*e.g.,* "AND", "OR", "XOR", *etc.*) to (one of) its operands; while "constraint" edges impose a constraint (*e.g.,* mandatory or optional) between nodes. Semantically, an FM specifies a set of **valid** products,

[1] The visual tree-like representation of feature models are feature diagrams. In the remaining of the thesis, feature models and feature diagrams will be used interchangeably since we are not interested on visualisation issues.

noted traditionally ⟦FM⟧: this set represents which (combinations of) features result in valid products or, alternatively, the sets of features, resulting in valid products, that satisfy the constraints expressed by the FM. Let us take a concrete example.

Example 5.1.1 (Feature Model for Soda Vending Machines)**.** Suppose we want to reason on a **family** of systems that share common functional features, but some products may have specific functionalities. In this case, we are interested in providing various institutions with Soda Vending Machines that are tailored to the needs of the institution's employees. For the sake of simplification, suppose that each institution offers either tea, soda or both beverages and that some institutions may offer free beverages. Vending machines accept payments in euros or dollars. In order to design and reason about a family of such systems, we need to define the characteristics of the vending machine hosted in each institution, captured by an FM.

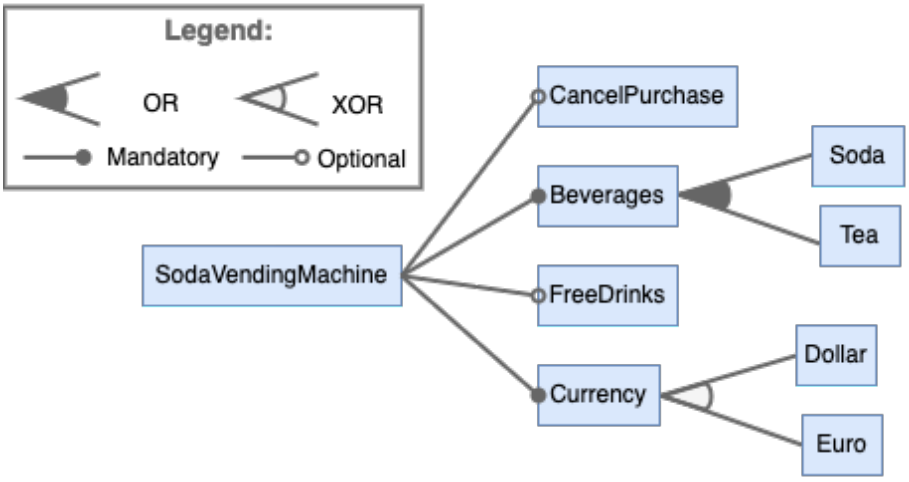

Figure 5.1: Feature model $\mathsf{FM}_{\mathsf{SVM}}(\mathsf{F}_{\mathsf{SVM}})$ of a soda vending machine, relying on the set of features $\mathsf{F}_{\mathsf{SVM}}$ [46, 61].

Figure 5.1 depicts an FM, named $\mathsf{FM}_{\mathsf{SVM}}(\mathsf{F}_{\mathsf{SVM}})$, for a family of Soda Vending Machines (SVM) based on a set of features encoded $\mathsf{F}_{\mathsf{SVM}}$. The family of soda vending machines provides two types of beverages (soda or tea), with the possibility to cancel an ongoing purchase. Optionally, it is possible to obtain free drinks. The vending machines can accept either euros or dollars. $\mathsf{FM}_{\mathsf{SVM}}(\mathsf{F}_{\mathsf{VM}})$ is composed of six concrete features, and three abstract features. Node Beverages is associated with an OR operator (depicted by an empty curved zone between the edges): it indicates that a given vending machine product may provide one (or both) beverage options. Node Currency is associated with a XOR operator (visually depicted by plain curved zone between the edges): it indicates that for a given vending machine product, one and only one currency must be selected. When clear from context, we drop the subscripts associated with features' and feature models' names: we would simply

note FM the feature model for the soda vending machines. $\mathsf{FM}_{\mathsf{SVM}}(\mathsf{F}_{\mathsf{SVM}})$ defines 24 different, valid products (*i.e.,* $|[\![\mathsf{FM}_{\mathsf{SVM}}(\mathsf{F}_{\mathsf{SVM}})]\!]| = 24$).

FMs have already been equipped with formal semantics [45,53,128,159,185,186], allowing various analyses (*e.g.,* consistency checking and counting *cf.* [24] for a survey on available analyses and related techniques). In this thesis, we assume that the FM of the system is already provided. Moreover, many approaches have already been defined for learning FMs, as previously stated (see Section 1.2). This assumption allows us to focus solely on behaviour.

5.2 On Featured Transition Systems' Infiniteness

Complementary to FM, we can compactly represent the behaviour of a VIS with a **Featured Transition System** (FTS) [48, 51]. An FTS captures the behaviour of an entire family of products (therefore relying on a feature set F) in a compact and concise way. To do so, they take advantage of their shared behaviour, specified by a set A of actions that the products may perform. Syntactically, FTSs extend the classical formalism of Transition Systems [20]: a transition describes an execution step between states, triggered by an external event that shall match the transition's label [81]. For FTSs, transitions are additionally annotated with **feature expressions** (*i.e.,* logical formulae over F) that denote the exact subset of features appearing in the valid products that may execute it.

Definition 3 (Feature Expression)**.** Let $\mathsf{F} = \{f_1, \ldots, f_{|\mathsf{F}|}\}$ be a set of features and associated to an FM. We denote by $\mathbb{B}(\mathsf{F})$ the set of propositional logic formulæ obtained from features as propositions, and with the classical Boolean operators. Each variable corresponds to a unique element of F whose semantics is a function $2^F \to \{\bot\ (\texttt{false}), \top\ (\texttt{true})\}$.

Each feature expression $\mathsf{b} \in \mathbb{B}(\mathsf{F})$ semantically represent **a subset of products** $\mathcal{P} \subseteq [\![\mathsf{FM}]\!]$. We say that $\mathcal{P}$ **satisfies** b, noted $\mathcal{P} \models \mathsf{b}$, when:

$$b = \bigvee_{p \in \mathcal{P}} \Big(\bigwedge_{f \in p} f \wedge \bigwedge_{g \in F \setminus p} \neg g \Big).$$

Before giving a formal definition of an FTS, let us take our previous Vending Machines example, whose FM (*i.e.,* its structural variability) is presented in Figure 5.1.

Example 5.2.1 (A Soda Vending Machine's FTS)**.** As expected, all products in a product line behave globally the same (*e.g.,* all soda vending machines deliver a beverage to a customer). Still, each particular vending machine may have a slightly different behaviour due to its specific combination of features. For example, a vending machine providing free beverages requires no payment before selecting a beverage.

Now, we can specify the behaviour of the soda vending machine hosted in each institution through an FTS, as presented in Figure 5.2. As it can be seen from the feature expressions, only specific configurations can execute some transitions:

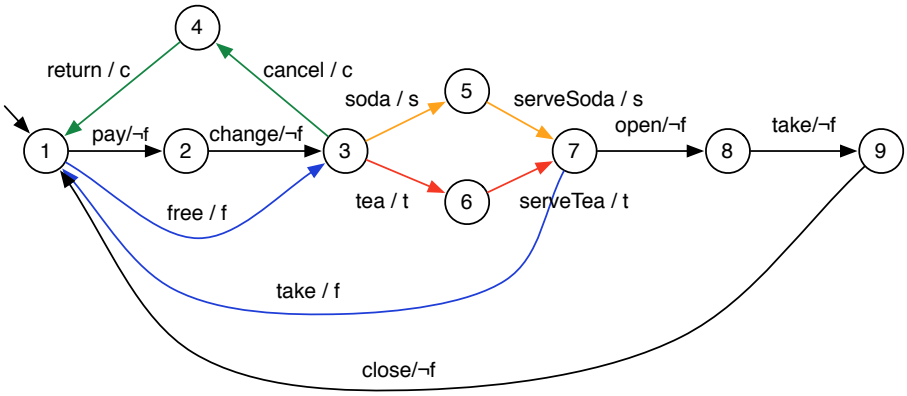

Figure 5.2: Featured Transition System for the Soda Vending Machines [46, 61].

e.g., only the vending machines with the free (`f`) option enabled can execute the `free` transition from state 1 to state 3.

Our initial goal was to learn FTSs according to their original definition by Cordy *et al.* [48, 51]. However, our primary source of information is **finite execution traces**, while TS [20] and FTS are representing infinite behaviour. This tension motivates us to propose a new formalism, more adequate for representing finite behaviour. In the soda vending machine example (see Figure 5.2), the `return`, `take` (the one indicated in blue) and `close` transitions mimics a **reset** of the system. More precisely, when we take those transitions, we need to "forget" the feature expressions, as if we had shut the system down and restarted it. For convenience, we address this by learning final states. This solution does not require simulating any reset of the system. In fact, this is not really a hard constraint since it is always possible to add empty transitions from final states back to the initial state **afterwards**, thereby restoring the infinite (F)TS. For this reason, the subsequent section will introduce a novel type of behavioural model, based on automata rather than transition systems. The definition of **featured deterministic finite automaton (FDFA)** is very similar to the classical FTS definition [48, 51], but has finite semantics.

5.3 Featured Deterministic Finite Automaton

Definition 4 recalls the typical definition of a Deterministic Finite Automaton (DFA).

Definition 4 (Deterministic Finite Automaton)**.** A deterministic finite automaton (DFA) is a tuple (A, Q, q_0, F, δ) where:

- A is a set of **actions**, also called the **alphabet**;
- Q is a finite set of states named the **state space**;
- $q_0 \in Q$ is **the initial state**;
- $F \subseteq Q$ is a set of **accepting (or final) states**;
- $\delta : Q \times \mathrm{A} \longrightarrow Q$ is the **transition function**.

Note that δ being a function, this definition ensures that the automaton is deterministic: for a given state and action, the targeted state is unique.

Now that we have defined them, we can combine FM, feature expressions and DFA to finally define the cornerstone of the thesis: **Featured Deterministic Finite Automaton (FDFA)**. An FDFA models the behaviour of a complete family of software products in one single model.

Definition 5 (Featured Deterministic Finite Automaton)**.** A featured deterministic finite automaton is a tuple $\mathsf{FDFA} = (\mathsf{FM}, \mathsf{A}, Q, q_0, F, \delta)$[2] where

- FM is a **feature model** over a set of features F;
- A is a set of **actions**, also called the **alphabet**;
- Q is a finite set of states named the **state space**;
- $q_0 \in Q$ is **the initial state**;
- $F \subseteq Q$ is a set of **accepting (or final) states**;
- $\delta : Q \times \mathsf{A} \longrightarrow \mathbb{B}(\mathsf{F}) \times Q$ is the **transition function**, where $\delta(q, \alpha) = (\mathsf{f}, q')$ (usually noted $q \xrightarrow{\alpha\ [\mathsf{f}]} q'$) means that there is a transition from state q to state q' labelled with action α and guarded with a feature expression F over F.

The guarding feature expression defines which subset of products is allowed to perform a given action in a certain state.

To derive the DFA for a specific product, the FDFA is **projected** on this product by pruning the transitions whose feature expressions are not satisfied and by removing all feature expressions. This process is analogous to the one performed on FTS to obtain the TS of a product [48, 51].

Formally, the projection operator is defined as follows:

Definition 6 (Projection operator)**.** Let $\mathsf{FDFA} = (\mathsf{FM}, \mathsf{A}, Q, q_0, F, \delta)$ be an FDFA, and $p \in [\![\mathsf{FM}]\!]$ be a product of the feature model FM. The projection of FDFA onto p, denoted $\mathsf{FDFA}_{|p}$, is the $DFA(\mathsf{A}, Q, q_0, F, \delta')$ where

$$\delta' = \left\{ q \xrightarrow{\alpha\ [\mathsf{f}]} q' \in \delta \;\middle|\; p \models \mathsf{f} \right\}.$$

We require all transitions of an FDFA to be deterministic *w.r.t.* the features, in the sense that if we take the projected automaton of each of the n products, we obtain n deterministic automaton. We define this determinism in Definition 7.

Definition 7 (Featured determinism)**.** A transition in a $\mathsf{FDFA} = (\mathsf{FM}, \mathsf{A}, Q, q_0, F, \delta)$ is said deterministic if there is only one possible output transition for an action and a specific configuration in each state:

$$\forall (q, \alpha, f, q'),\ (q, \alpha, f', q'') \in \delta : [\![f]\!] \cap [\![f']\!] \neq \emptyset \Rightarrow q' = q''.$$

The non-empty intersection ensures that if at least one product exists satisfying both f and f', then both transitions have the same target.

[2] In the original definition of FTS [48, 51], they also define a labelling function that associates every state with the set of atomic propositions satisfied by this state. In our context, this function is not needed but can be computed by walking through the graph and accumulating feature expressions in each state.

As for simple DFA, determinism is ensured by the uniqueness of the targeted state in the δ function.

We denote by $\mathcal{L}(\mathsf{FDFA})$ the language accepted by the FDFA automaton. In other words, $\mathcal{L}(\mathsf{FDFA})$ is the set of all **accepting runs** (Definition 8) on FDFA. A word of $\mathcal{L}$ is thus a sequence of actions from A combined with a feature expression on F.

Definition 8 (Run)**.** A **run** on an $\mathsf{FDFA} = (\mathsf{FM}, \mathsf{A}, Q, q_0, F, \delta)$ is a non-empty and finite sequence of transitions, beginning from the initial state:

$$q_0 \xrightarrow{\alpha_1\ [f_1]} q_1 \xrightarrow{\alpha_2\ [f_2]} \ldots \xrightarrow{\alpha_{n-1}\ [f_{n-1}]} q_{n-1} \xrightarrow{\alpha_n\ [f_n]} q_n$$

Where $q_0, q_1, ..., q_{n-1}, q_n \in Q$. Only the products satisfying all the feature expressions $(f_1, ..., f_n) \in \mathbb{B}(\mathsf{F})$ can execute this run. A run is called **accepting** if it ends in a finite state ($f_n \in F$). If not, the run is **rejecting**.

Definition 9 (Featured language)**.** The language $\mathcal{L}(\mathsf{FDFA})$ defined by FDFA can be seen as the union of all the languages accepted by each projected TS:

$$\mathcal{L}(\mathsf{FDFA}) = \bigcup_{p \in [\![\mathsf{FM}]\!]} \mathcal{L}\left(\mathsf{FDFA}_{|p}\right).$$

To highlight this union, we can refer to $\mathcal{L}(\mathsf{FDFA})$ as a **featured language**.

Definition 10 (Word)**.** For each accepting run $q_0 \xrightarrow{\alpha_1\ [f_1]} \ldots \xrightarrow{\alpha_n\ [f_n]} q_n$ over FDFA, we can define a **word** of the language $\mathcal{L}(\mathsf{FDFA})$. A word w is a sequence of actions associated with a feature expression:

$$w = \alpha_1 \cdot \alpha_2 \cdot \ldots \cdot \alpha_n\ [f] \quad \in \mathcal{L}(\mathsf{FDFA})$$

where $f = f_1 \wedge f_2 \wedge \ldots \wedge f_{n-1} \wedge f_n$ and $\alpha_i \cdot \alpha_j$ denotes the concatenation operation over A. This operator is often omitted when non-ambiguous and clear from the context. Moreover, $\mathsf{Seq}(\mathsf{A}^\star)$ is a sequence of words from $\mathsf{A}^\star$ and $w_1 \oplus w_2$ denotes the concatenation of the two words w_1 and w_2.

5.4 L^* Algorithm

According to Frits Vaandrager [216], the problem of inferring behaviours from software artefacts is not new. Back in 1956, Moore [161] already proposed the first approach to infer finite machines. Since then, the field of behavioural inference and more precisely **automata learning** has kept on expanding. One major contribution in this area remains the L^* algorithm proposed by Dana Angluin in 1987 [9].

The L^* algorithm is defined as a **black-box** approach, meaning that it learns a model of a system exclusively from execution traces (*i.e.,* without requiring any access to its source code). The traces are obtained via **active** interaction with the **System Under Learning (SUL)**. This algorithm aims at actively learning the automaton of an unknown regular language, a finite state machine (FSM) for example. The main idea is an abstraction of the learning process where a **Learner** tries to progressively build a model of the SUL by sending two types of queries to a **Teacher** (as shown

in Figure 5.3). It uses **Membership Queries (MQ)** to know if a sequence (a word) belongs to the language to learn. Once the Learner found a valid hypothesis model (as defined in Section 5.4.2), it uses **Equivalence Queries (EQ)** to ask the Teacher if it is Equivalent to the SUL. If not, the Teacher sends a **counterexample** (*i.e.*, a word that is in the language but not accepted by the hypothesis automaton, or vice versa). Then, the Learner can use this counterexample to explore a new part of the SUL and repeat the whole process until equivalence is achieved. The LearnLib framework [158], widely used in the research community, implements these concepts. More recently, Muškardin *et al.* [163] proposed another implementation named AALpy.

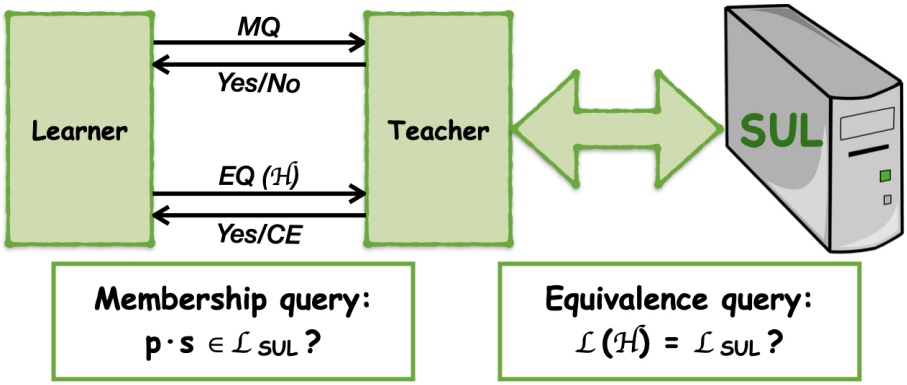

Figure 5.3: Classical L^* algorithm

L^* can be divided into three phases, summarised as follows:

(i) **The observation table ($\mathcal{OT}$) construction** where the learner sends membership queries and builds an Observation Table, based on the Teacher's answers;
(ii) **The hypothesis validation** where the learner constructs a hypothesis automaton from the Observation Table and sends it to the teacher for equivalence query;
(iii) **The counterexample analysis** where the learner refines the hypothesis, by analysing the counterexample.

This process is iterative and monotonically adds states and transitions to the hypothesis automata until the Teacher is no longer able to provide a counterexample. We detail these three phases below, referring to lines in Algorithm 1. In parallel, we show the different steps in an example based on the soda vending machine depicted in the previous sections.

Example 5.4.1 (Product selection)**.** The original version of L^* algorithm allows us to learn a single product. To show how the algorithm work, we thus need to select this product from the FM presented in Figure 5.1. We have chosen the vending machine with the options $CancelPurchase$, $Euro$ and $Soda$, activated. The other unnecessary features are deactivated. By projection of the FTS presented in Figure 5.2, we obtain the behaviour depicted by Figure 5.4.

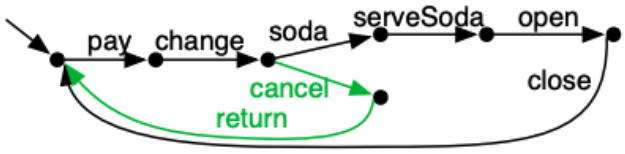

Figure 5.4: Transition system of a specific vending machine obtained through projection.

1 $P \leftarrow \{\epsilon\}$;
2 $S \leftarrow \{\epsilon\}$;
3 $P^+ \leftarrow \{\}$;
4 $T[\epsilon,\epsilon] \leftarrow \mathcal{MQ}(\epsilon,\epsilon)$;
} ▷ $\mathcal{OT}$ initialisation
5 **while** ∃ ce **do**
6 | **while** !closed ∨ !consistent **do**
7 | | $P^+ \leftarrow P^+ \cup P \cdot A$; ▷ $\mathcal{OT}$ Extension
8 | | $\forall p \in P, \forall s \in S,\ T[p,s] \leftarrow \mathcal{MQ}(p,s)$;
9 | | $\forall p^+ \in P^+, \forall s \in S,\ T^+[p^+,s] \leftarrow \mathcal{MQ}(p^+,s)$;
} ▷ Filling $\mathcal{OT}$
10 | |
11 | | **if** !closed **then**
12 | | | Move extended prefixes to (closed) prefixes;
13 | | **end**
14 | | **if** !consistent **then**
15 | | | Add new suffixes;
16 | | **end**
} ▷ Making $\mathcal{OT}$ closed and consistent
17 | |
18 | **end**
19 | $\mathcal{H} \leftarrow MakeConjecture(T, T^+)$; ▷ Building Hypothesis automaton
20 | ce $\leftarrow \mathcal{EQ}(\mathcal{H})$;
21 | $P \leftarrow P \cup$ prefixes(ce);
} ▷ $\mathcal{EQ}$ and counterexample analysis
22 |
23 **end**
24 **return** $\mathcal{H}$

Algorithm 1: Pseudo-code of L^* algorithm

5.4.1 Building an Observation Table

An **Observation Table** is an intermediate data structure which stores the answers to the membership queries. The hypothesis automaton is built upon the observation table. It is made of sequences of actions called prefixes and suffixes. Prefixes label table rows while suffixes are labels of columns. When we combine a prefix with a suffix, we obtain a word. If the membership query for this word is positive, then the word belongs to the language described by the SUL. $T[p,s]$ represents the cell of the table corresponding to prefix p and suffix s. The cell contains `true` if the word $w = p \cdot s$ is accepted by the SUL and `false` otherwise.

The observation table can be divided into two parts, defined on the same set S of **suffixes**. The upper part registers a set P of **closed prefixes**, while the lower part registers **extended prefixes**, noted P^+. An extended prefix p^+ is a one-step extension of a closed prefix p: $p^+ = p \cdot \alpha$ where $\alpha \in A$ (alphabet of actions from the SUL). To denote the set of all values for a specific prefix (*w.r.t.* each suffix), we use rows.

Definition 11 (Row)**.** Let p be a closed-prefix in $P \bigcup P^+$,

$$row(p) = \{\mathcal{MQ}(p,s) | s \in S\}$$

In the algorithm (see Algorithm 1), `lines 1 – 3` initialise the $\mathcal{OT}$ with the silent prefix, the silent suffix and an empty set of extended prefixes (as shown in Example 5.4.2). Then, we start the iterative process described earlier. `Lines 5 – 7` extends the table by adding new extended prefixes and filling the cells with associated $\mathcal{MQ}$ (Example 5.4.3).

Example 5.4.2 (Table Initialisation)**.** After initialisation, we obtain Table 5.1.

Table 5.1: Initialisation of an Observation Table

OT		**S**	
		ϵ	
P	ϵ		
P⁺			

Example 5.4.3 (Table Extension)**.** Since we only have the silent prefix in the table, we can extend it by adding each action from the alphabet to the list of extended prefixes, as in Table 5.2.

Then, we can fill the cells with the result of the membership queries to obtain Table 5.3. For this example, each query returns `false` because a single action suffixed by the silent prefix is not a complete valid word: $\mathcal{MQ}(pay,\epsilon) \rightarrow$ `false`

5.4.2 Hypothesis Construction & Validation

A hypothesis automaton $\mathcal{H}$ is a candidate automaton build from the Observation Table once it is **closed** and **consistent**. The hypothesis is sent to the teacher and is

Table 5.2: Initialisation of an Observation Table

OT		S	
		ϵ	
P	ϵ		
P$^+$	*pay*		
	change		
	cancel		
	return		
	...		

Table 5.3: OT Filling

OT		S	
		ϵ	
P	ϵ	`false`	
P$^+$	*pay*	`false`	
	change	`false`	
	cancel	`false`	
	return	`false`	
	...	...	

compared to the SUL through an equivalence query (Algorithm 1, `lines 19 – 20`). Definitions 12 and 13 respectively define the closedness and consistency properties. Examples 5.4.4 and 5.4.5 show how to transform the table to make it closed and consistent.

Definition 12 (Closedness)**.** An $\mathcal{OT}$ is closed if and only if:

$$\forall p^+ \in P^+, \exists p \in P : row(p^+) = row(p)$$

Conceptually, for each row in the lower part of the $\mathcal{OT}$, there exists an equivalent row in the upper part. If the $\mathcal{OT}$ is not closed, we need to move p^+ from P^+ to P. Closedness ensures that all transitions have a target location.

Example 5.4.4 (Closing the Table)**.** The last row in Table 5.4 was not matching the unique row of the upper table: one accepted the ϵ suffix, the other did not. To close the table, we move this prefix from the lower to the upper part of the table. Now, every extended prefix has a match in the upper part.

Definition 13 (Consistency)**.** An $\mathcal{OT}$ is consistent if and only if:

$$\forall p_1, p_2 \in P,\ row(p_1) = row(p_2) \Rightarrow \forall \alpha \in A :\ row(p_1 \cdot \alpha) = row(p_2 \cdot \alpha)$$

If the $\mathcal{OT}$ is not consistent, we need to add $\alpha \cdot s$ to S. Consistency ensures that whenever a product accepts the prefix $p \cdot \alpha$, it also accepts the prefix p. Note that

Table 5.4: OT closing

OT		S	
		ϵ	
P	ϵ	false	
	$pay \cdot change \cdot cancel \cdot return$	true	
P⁺	pay	false	
	$change$	false	
	$cancel$	false	
	$return$	false	
	~~$pay \cdot change \cdot cancel \cdot return$~~	~~true~~	
	...	...	

such a definition of consistency is inline with the notion of determinism as defined in 7.

Example 5.4.5 (Making the Table Consistent)**.** in Table 5.5, we find two different rows (framed in blue and green respectively) which have the same value for each suffix. However, when we look one $return$ further (prefixes highlighted by an arrow of the right colour), we obtain a different value for the suffix ϵ. This is an inconsistency.

Table 5.5: OT consistency

OT		S	
		ϵ	
P	ϵ	false	
	⇒$pay \cdot change \cdot cancel \cdot return$	true	
	$pay \cdot change \cdot cancel$	false	
	...	...	
P⁺	pay	false	
	$change$	false	
	$cancel$	false	
	⇒$return$	false	
	...	...	

To solve it, we need to add $return$ as a new suffix and fill the table with the corresponding membership queries. Table 5.6 is now consistent: the two problematic rows are differentiated.

To build the automaton from the $\mathcal{OT}$, each prefix is transformed into an automaton state and suffixes define the next transitions from each state, like in Example 5.4.6.

Example 5.4.6 (Building Hypothesis)**.** From the closed and consistent Table 5.7, we

Table 5.6: OT consistency

OT		S	
		ϵ	$return$
P	ϵ	false	false
	$pay \cdot change \cdot cancel \cdot return$	true	false
	$pay \cdot change \cdot cancel$	false	true
	...	...	...
P$^+$	pay	false	false
	$change$	false	false
	$cancel$	false	false
	$return$	false	false
	...	...	...

Table 5.7: Building hypothesis from OT

OT		S		
		ϵ	$return$	**...**
P	ϵ	false	false	...
	pay	false	false	...
	$pay \cdot change$	false	false	...
	$pay \cdot change \cdot cancel$	false	true	...
	$pay \cdot change \cdot cancel \cdot return$	true	false	...
	...	...	...	...
P$^+$	...	...	...	...

can define a new DFA as following:

$$DFA(A, Q, q_0, F, \delta): \tag{5.1}$$
$$A = \{pay, change, ...\} \tag{5.2}$$
$$Q = \{q_\epsilon, q_{pay}, q_{pay \cdot change}, ...\} \tag{5.3}$$
$$q_0 = q_\epsilon \tag{5.4}$$
$$F = \{q_{pay \cdot change \cdot cancel \cdot return}\} \tag{5.5}$$
$$\delta(q_\epsilon, pay) = q_{pay},\ \delta(q_{pay}, change) = q_{pay \cdot change}, ... \tag{5.6}$$

5.4.3 Counterexample Analysis & $\mathcal{OT}$ Refinement

By sending an equivalence query to the Teacher, the Learner asks if the hypothesis automaton $\mathcal{H}$ is equivalent to the SUL. If the hypothesis is considered equivalent, the Teacher terminates the execution and returns $\mathcal{H}$ as an adequate model for the SUL. Otherwise, the Learner receives a counterexample. The counterexample is a sequence of actions that are not accepted by $\mathcal{H}$ whereas accepted by the SUL or vice-versa.

Definition 14 (Counterexample)**.** A counterexample is a word which is accepted

(resp. rejected) by the $\mathcal{H}$ hypothesis but rejected (resp. accepted) by the system under learning.

If the equivalence query returns a counterexample, the learner has to adapt the Observation Table in consequence, as shown in Example 5.4.7. Counterexamples allow the algorithm to explore new parts of the language by adding new prefixes (Algorithm 1, `line 21`). These changes concretely impact the hypothesis automaton by adding new states and transitions.

Example 5.4.7 (Counterexample Analysis)**.** Suppose the equivalence query returns the following counterexample: $pay \cdot change \cdot cancel \cdot return$. We need to add four new prefixes in our table, resulting in Table 5.8. After this step, we can pursue the learning by extending the table and making it closed and consistent again.

Table 5.8: Updating OT with counterexample

OT		**S**	
		ϵ	
P	ϵ	`false`	
	pay	`false`	
	$pay \cdot change$	`false`	
	$pay \cdot change \cdot cancel$	`false`	
	$pay \cdot change \cdot cancel \cdot return$	`true`	
$\mathbf{P}^+$			

CHAPTER 6

FEATURED-L* SPECIFICATION

To answer **RQ$_1$**, we chose to adapt L^* to take variability into account. Unlike previous approaches (*e.g.,* [54, 56, 63, 201]), we aim to treat a feature expression as a first-class citizen during learning. Our new algorithm, called **Featured-L^* (FL^*)**, aims to build an FDFA for some unknown featured language, with a limited number of queries to the SUL.

As for L^* and all active learning approaches, FL^* requires access to the system to be able to execute sequences of actions or a simulator of the system (*e.g.,* another behavioural model). It is also assumed that the Feature Model of the system is given. This assumption ensures that we consider only **valid** products.

Figure 6.1 illustrates the additions we proposed to the classical L^* algorithm. To learn an **FDFA**, we need to infer transitions, states and feature expressions over transitions. FL^* follows the same 3-phase iterative structure as L^*: hypothesis construction, hypothesis validation, and counterexample processing. A complete example of learning, displaying all the important data structures can be found in Appendix A.

In classical active learning algorithms such as L^*, prefixes are used to identify potential states. The Learner sends membership queries to the Teacher and compares the answers to determine when prefixes should correspond to the same state. If two prefixes have equivalent rows, meaning they exhibit the same behaviour for all possible suffixes, they are either both accepted or both rejected by the SUL. In this case, the prefixes are considered isomorphic and can be merged into a single state, resulting in a more compact representation of the system's behaviour. This approach avoids redundancy and improves the efficiency of the learning process.

In an FDFA, each state can still be identified by a (featured) prefix, but things are a bit more complicated to decide if two prefixes lead to the same state. Adding the same suffix and comparing the two words' behaviour is not sufficient for this task.

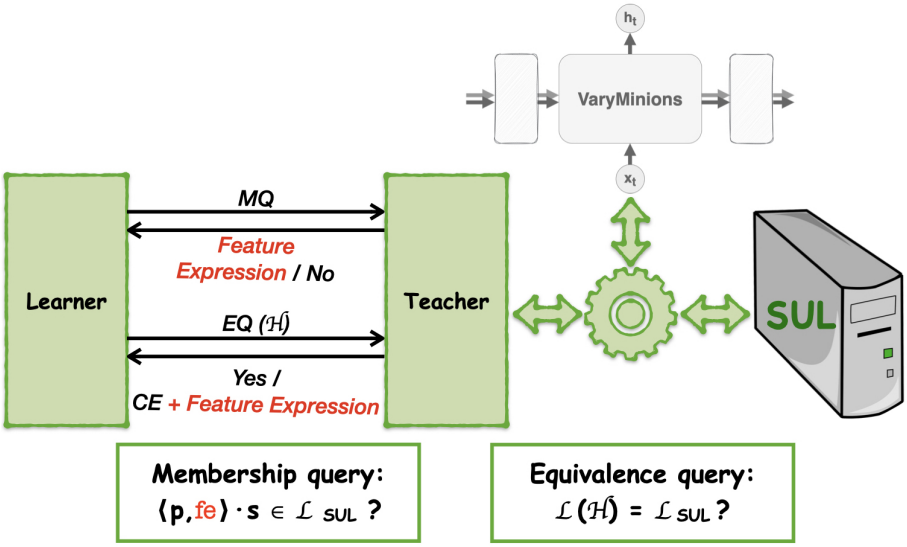

Figure 6.1: FL^* algorithm, with adaptations from classical $\mathbf{L}^*$ in red

Two equivalent prefixes could describe the behaviour of disjoint sets of products and thereby describe different states. To solve this issue, we aim to take feature expressions into account, specifying the right subset of product for each prefix. We adapt *membership queries* to deal with feature expressions. Thereby, instead of answering a simple yes or no, membership queries will provide a feature expression [1] representing the subset of products accepting each word. It also associates each counterexample with a feature expression [1], providing the list of products for which behaviour do not match: behaviour is either accepted by the hypothesis or the SUL, but not both.

The remaining of this chapter is dedicated to the description of FL^*, our new algorithm that learns an FDFA for a whole SPL. Algorithm 2 presents the general structure of the algorithm, that we refer to when we detail each step. We exemplify each step with the soda vending machine presenting earlier (see FM in Figure 5.1 and FTS in Figure 5.2). Then, we use these concepts and definitions to provide a complete version of the algorithm. To conclude this chapter, we compare our approach with three related state-of-the-art algorithms.

6.1 Learning the Observation Table

As we have seen, an observation table is the key data structure to build a hypothesis automaton. An observation table is a double-entry table. An $\mathcal{OT}$ cell is defined by both a prefix key and a suffix key. While in L^* we stored `true` or `false` values, here

[1] The hypothesis that the Teacher is able to relate queries with structural variability is non-trivial and its implications will be further discussed in Section 9.3.

```
1  P ← {⟨ϵ, FM⟩};                                          ▷ OT initialisation
2  S ← {ϵ};
3  P⁺ ← {};
4  T[⟨ϵ, FM⟩, ϵ] ← MQ(⟨ϵ, FM⟩, ϵ);
5    while ∃ (ce, fe) do
6    | while !closed ∨ !consistent do
7    |   P⁺ ← P⁺ ∪ P · A;                                  ▷ OT Extension
8    |   ∀p ∈ P, ∀s ∈ S, T[p, s] ← MQ(p, s);               ▷ Filling OT
9    |   ∀p⁺ ∈ P⁺, ∀s ∈ S, T⁺[p⁺, s] ← MQ(p⁺, s);
10   |
11   |   if !closed then                                   ▷ Making OT closed and
12   |   |   Move extended prefixes to (closed) prefixes;    consistent
13   |   end
14   |   if !consistent then
15   |   |   Add new suffixes;
16   |   end
17   |
18   | end
19   | H ← MakeConjecture(T, T⁺);   ▷ Building Hypothesis automaton
20   | (ce, fe) ← EQ(H);                                   ▷ EQ and counterexample
21   | P ← P ∪ {⟨p, fe⟩ | p ∈ prefixes(ce)};                  analysis
22   |
23 end
24 return H
```

Algorithm 2: General pseudo-code of FL^* algorithm

a cell contains a feature expression specifying which subset of products accept the word defined by the prefix and suffix keys.

In the context of FL^*, a prefix is always associated with a feature expression. Suffixes are "free", in the sense that they are not restricted to a subset of products. This idea was inspired by Cassel *et al.* [38,39] and *concolic testing* approaches [99,188] where"free" variables are opposed to "bounded" variables. The latter is a technique mixing concrete and symbolic execution. In concolic testing, we use a theorem prover and constraint logic programming to generate test cases that achieve good program coverage. The former is an application of L^* to Extended Finite State Machine learning. They define symbolic suffixes as *"a sequence of symbols with uninstantiated data parameters"* and Symbolic Decision Trees which models how data parameters in a given set of suffixes affect whether continuations of a given prefix should be accepted by the automaton or not. So, in this sense, prefixes are seen as a concrete, instantiated part of a word whose suffixes are the symbolic part.

Definition 15 (Featured Prefix)**.** A **featured prefix** $p \in P$ is a pair composed of a sequence of actions and a feature expression: $\langle seq, \mathtt{fe} \rangle \in \mathsf{Seq}(\mathsf{A}^\star) \times \mathbb{B}(\mathsf{F})$.

Definition 16 (Featured Extended Prefixes)**.** The set of **featured extended prefixes**

can be defined as:

$$P^+ = \{\langle seq \cdot \alpha, \texttt{fe}\rangle | \langle seq, \texttt{fe}\rangle \in P, \alpha \in \mathsf{A}^\star\}$$

For simplification, $\langle seq \cdot \alpha, \texttt{fe}\rangle$ can be replaced by $p \cdot \alpha$ when extending an existing featured prefix $p = \langle seq, \texttt{fe}\rangle \in P$.

Definition 17 (Suffix)**.** A **suffix** $s \in S$ is a sequence of actions: $\mathsf{Seq}(\mathsf{A}^\star)$.

Definitions 15, 17 and 16 respectively define simple featured prefixes, suffixes and featured extended prefixes. Definition 18 captures the notion of the observation table. Note that in the rest of this thesis, we will generally omit the term "featured" when we are talking about featured prefixes and extended featured prefixes.

Definition 18 (Observation table)**.** An **observation table** OT is a pair of functions that associates a feature expression to a pair of an (extended) prefix/suffix:

$$\mathsf{OT} = (\mathsf{T}, \mathsf{T}^+) \text{ with:} \tag{6.1}$$
$$\mathsf{T} = (P \times S) \longrightarrow \mathbb{B}(\mathsf{F}) \tag{6.2}$$
$$\mathsf{T}^+ = (P^+ \times S) \longrightarrow \mathbb{B}(\mathsf{F}) \tag{6.3}$$

T is usually called the upper part and T^+ the lower part of the table. The upper part contains short prefixes while the lower part contains extended prefixes. Note that T and T^+ share the same set S of suffixes. Since we only manipulate (partial) functions, we adopt an array-like notation: for $p \in P$ and $s \in S$, we note $\mathsf{T}[p, s]$ instead of $\mathsf{T}(p, s)$.

The function $\texttt{row} : P \cup P^+ \longrightarrow \mathbb{B}(\mathsf{F}) \times ... \times \mathbb{B}(\mathsf{F})$ returns the set of values contained by a specific row. By definition,

$$\forall p \in P \cup P^+, \texttt{row}(p) = \{\texttt{fe} | \mathsf{T}(p, s) = \texttt{fe},\ s \in S\}$$

In FL^*, we initialise the $\mathcal{OT}$ with a prefix composed of the empty sequence (denoted by the ϵ symbol) and the feature model, as shown in Example 6.1.1. The set of suffixes only contains the empty sequence and the set of extended prefixes is empty. Since we have one short prefix and one suffix, we can execute our first membership query to fill the only cell of $\mathcal{OT}$ (see Definition 19). We will repeat this operation each time we add an (extended) prefix or a suffix to the table (*i.e.,* each time we create more cells).

Example 6.1.1 (Table Initialisation)**.** After initialisation, we obtain Table 6.1. Changes form the original algorithm are highlighted in green.

Definition 19 ($\mathcal{MQ}$)**.** The function $\mathcal{MQ} : P \times S \longrightarrow \mathbb{B}(\mathsf{F})$ corresponds to a membership query sent by the Learner to the Teacher. Intuitively, the Learner ask if concatenating the prefix and the suffix defines a valid word (*i.e.,* accepted by the SUL):

$$w = \langle p, \texttt{fe}\rangle \cdot s \in L_{SUL}\,?$$

Table 6.1: Initialisation of an Observation Table

OT		S	
		ϵ	
P	$\langle \epsilon, \texttt{FM} \rangle$		
P$^+$			

The Teacher should answer by a formula ($\texttt{fe} \in \mathbb{B}(\mathsf{F})$) specifying the conditions under which the trace is accepted. If the word is invalid for all products, then it returns false.

Until the algorithm completes (*i.e.,* while there exists some counterexample), we loop to extend the table. First, we extend all short prefixes by adding one action to the short prefix, preserving the associated feature expression (as in Example 6.1.2). After extension, we check if the table respects two properties: **closedness** (Definition 20) and **consistency** (Definition 21). Compared to the classic version of L^*, closedness is equivalent (see Example 6.1.3). New featured (extended) prefixes are added to the table only if i) they are not already present and ii) they are not a combination of other prefixes in the table. This particularity ensures that closedness is "safe" regarding feature expressions and does not require any modification compare to its original definition. Nevertheless, consistency is modified to take feature expressions into account (Example 6.1.4). More specifically, when we add one symbol to a short prefix, it can lead to **more than one** extended prefix. To ensure that every product is considered, we need to consider the union of all those prefixes. We loop until both properties are respected.

Example 6.1.2 (Table Extension)**.** Since we only have the silent prefix in the table, we can extend it by adding each action from the alphabet to the list of extended prefixes, as in Table 6.2. We now have featured (extended) prefixes instead of classical prefixes. Note that when we combine all feature expressions (*i.e.,* by a disjunction) associated with one prefix, we should always obtain the FM. It means that if we add a prefix associated with $\texttt{f} \wedge \texttt{FM}$ to the table, we need to ensure that $\neg\texttt{f} \wedge \texttt{FM}$ is also in the table. This is very important to ensure feature determinism. At this stage of our example however, the only feature expression in the table is FM, whose negation $\texttt{FM}\neg\texttt{FM} = \texttt{false}$ can be ignored.

Then, we can fill the cells with the result of the membership queries to obtain Table 6.3. For this example, each query returns false because a single action suffixed by the silent prefix is not a complete valid word. In the general case, we can obtain any feature expression *w.r.t.* the FM.

Definition 20 (Closedness)**.** An $\mathcal{OT}$ is closed if and only if:

$$\forall p^+ \in P^+,\ \exists p \in P : \texttt{row}(p^+) = \texttt{row}(p)$$

Table 6.2: Initialisation of an Observation Table

OT		S	
		ϵ	
P	$\langle \epsilon, \texttt{FM} \rangle$		
P$^+$	$\langle pay, \texttt{FM} \rangle$		
	$\langle change, \texttt{FM} \rangle$		
	$\langle cancel, \texttt{FM} \rangle$		
	$\langle return, \texttt{FM} \rangle$		
	...		

Table 6.3: OT Filling

OT		S	
		ϵ	
P	$\langle \epsilon, \texttt{FM} \rangle$	false	
P$^+$	$\langle pay, \texttt{FM} \rangle$	false	
	$\langle change, \texttt{FM} \rangle$	false	
	$\langle cancel, \texttt{FM} \rangle$	false	
	$\langle return, \texttt{FM} \rangle$	false	
	...	...	

Conceptually, for each row in the lower part of the $\mathcal{OT}$, there exists an equivalent row in the upper part. If the $\mathcal{OT}$ is not closed, we need to move p^+ from P^+ to P. Closedness ensures that all transitions have a target location.

Example 6.1.3 (Closing the Table)**.** To close Table 6.4, we move the prefix $\langle pay \cdot change \cdot cancel \cdot return, \texttt{C} \rangle$ from the lower to the upper part of the table. Now, every extended prefix has a match in the upper part. Note that for readability purpose we omit the restriction to the FM in the table (*e.g.*, C instead of$\texttt{C} \wedge \texttt{FM}$).

Table 6.4: OT closing

OT		S	
		ϵ	
P	$\langle \epsilon, \texttt{FM} \rangle$	false	
	$\langle pay \cdot change \cdot cancel \cdot return, \texttt{C} \rangle$	C	
P$^+$	$\langle pay, \texttt{FM} \rangle$	false	
	$\langle change, \texttt{FM} \rangle$	false	
	$\langle cancel, \texttt{FM} \rangle$	false	
	$\langle return, \texttt{FM} \rangle$	false	
	~~$\langle pay \cdot change \cdot cancel \cdot return, \texttt{C} \rangle$~~	~~C~~	
	$\langle pay \cdot change \cdot cancel \cdot return, \neg \texttt{C} \rangle$	false	
	...	...	

Definition 21 (Consistency)**.** An $\mathcal{OT}$ is consistent if and only if:

$$\forall p_1, p_2 \in P \cup P^+ \text{ such that } p_1 = \langle seq_1, \texttt{fe}_1 \rangle,\ p_2 = \langle seq_2, \texttt{fe}_2 \rangle :$$
$$\texttt{row}(p_1) = \texttt{row}(p_2) \Rightarrow \forall s \in S,\ \forall \alpha \in A,\ \bigvee_{p \in \mathcal{C}_1} T[p, s] = \bigvee_{p \in \mathcal{C}_2} T[p, s]$$

where

$$\mathcal{C}_1 = \{p \in P \cup P^+ | p = \langle seq_1 \cdot \alpha, \texttt{fe} \rangle,\ \texttt{fe} \in \mathbb{B}(\mathsf{F})\}$$
$$\mathcal{C}_2 = \{p \in P \cup P^+ | p = \langle seq_2 \cdot \alpha, \texttt{fe} \rangle,\ \texttt{fe} \in \mathbb{B}(\mathsf{F})\}$$

If the $\mathcal{OT}$ is not consistent, we need to add $\alpha \cdot s$ to S. Consistency ensures that whenever a product accepts the prefix $p \cdot \alpha$, it also accepts the prefix p.

Table 6.5: OT consistency

	OT	**S**
		ϵ
P	$\langle \epsilon, \mathsf{FM} \rangle$	false
	$\Rightarrow \langle pay \cdot change \cdot cancel \cdot return, \mathsf{C} \rangle$	C
	$\langle pay \cdot change \cdot cancel, \mathsf{C} \rangle$	false
	...	...
P$^+$	$\langle pay, \mathsf{FM} \rangle$	false
	$\langle change, \mathsf{FM} \rangle$	false
	$\langle cancel, \mathsf{FM} \rangle$	false
	$\Rightarrow \langle return, \mathsf{FM} \rangle$	false
	$\Rightarrow \langle pay \cdot change \cdot cancel \cdot return, \neg \mathsf{C} \rangle$	false
	...	...

Example 6.1.4 (Making the Table Consistent)**.** Applying Definition 21 on Table 6.5, we can find p_1 in blue and p_2 in green whose rows are equals. Then, we can look one $return$ further and compare all matching prefixes. For p_1, we obtain only one extended prefix, highlighted by the blue arrow. For p_2, we obtain two prefixes with different feature expressions, as shown by green arrows. Thus, we obtain:

$$\mathcal{C}_1 = \{\langle return, \mathsf{FM} \rangle\}$$
$$\mathcal{C}_2 = \{\langle pay \cdot change \cdot cancel \cdot return, \mathsf{C} \rangle, \langle pay \cdot change \cdot cancel \cdot return, \neg \mathsf{C} \rangle\}$$

Since we have one single suffix, we have one single comparison to make:

$$T[\langle return, \mathsf{FM} \rangle, \epsilon] = \texttt{false} \neq \bigvee_{p \in \mathcal{C}_2} T[p, \epsilon] = (\mathsf{C} \vee \texttt{false}) = C$$

.

To solve this inconsistency, we need to add $return$ as a new suffix and fill the table with the corresponding membership queries. Table 6.6 is now consistent: the two problematic rows are differentiated.

Table 6.6: OT consistency

OT		S	
		ϵ	$return$
P	$\langle \epsilon, \texttt{FM} \rangle$	false	false
	$\langle pay \cdot change \cdot cancel \cdot return, \texttt{C} \rangle$	C	false
	$\langle pay \cdot change \cdot cancel, \texttt{C} \rangle$	false	C
	...	...	...
P$^+$	$\langle pay, \texttt{FM} \rangle$	false	false
	$\langle change, \texttt{FM} \rangle$	false	false
	$\langle cancel, \texttt{FM} \rangle$	false	false
	$\langle return, \texttt{FM} \rangle$	false	false
	$\langle pay \cdot change \cdot cancel \cdot return, \neg \texttt{C} \rangle$	false	false
	...	...	...

6.2 Hypothesis Construction and Validation

A hypothesis automaton $\mathcal{H}$ is a candidate FDFA build from the Observation Table once it is both **closed** and **consistent**. Algorithm 3 details how a hypothesis FDFA $\mathcal{H}$ is created from the observation table. Example 6.2.1 highlight this procedure. Each state of the automaton ($q_p \in Q$) is defined by a minimal prefix ($p \in P$). This procedure depends on the `Targets` function, which defines the set of targeted states [2] for a transition from a specific state and with specific actions:

$$\texttt{Targets} : P \times A \longrightarrow P \quad \text{defined by:}$$

$$\texttt{Targets}(\langle seq, \texttt{fe} \rangle, \alpha) = \{\langle seq', \texttt{fe}' \rangle \in P \mid seq' = seq \cdot \alpha, \models (\texttt{fe}' \wedge \texttt{fe}) \neq \texttt{false}\} \cup \left\{ p \in P \mid \exists \langle seq', \texttt{fe}' \rangle \in P^+, \texttt{row}(p) = \texttt{row}(seq', \texttt{fe}'), \; seq' = seq \cdot \alpha, \models (\texttt{fe}' \wedge \texttt{fe}) \neq \texttt{false} \right\}$$

Example 6.2.1 (Building Hypothesis)**.** From the closed and consistent Table 6.7, we can define a new FDFA as following:

$$FDFA(FM, A, Q, q_0, F, \delta): \tag{6.4}$$

$$FM = FM_{SVM} \tag{6.5}$$

$$A = \{pay, change, ...\} \tag{6.6}$$

$$Q = \{q_{\langle \epsilon, \texttt{FM} \rangle}, q_{\langle pay, \texttt{C} \rangle}, q_{pay \cdot change}, ...\} \tag{6.7}$$

$$q_0 = q_{\langle \epsilon, \texttt{FM} \rangle} \tag{6.8}$$

$$F = \{q_{\langle pay \cdot change \cdot cancel \cdot return, \texttt{C} \rangle}\} \tag{6.9}$$

$$\delta(q_{\langle \epsilon, \texttt{FM} \rangle}, pay) = (\texttt{C}, q_{\langle pay, \texttt{C} \rangle}),$$
$$\delta(q_{\langle pay, \texttt{C} \rangle}, change) = (\texttt{C}, q_{\langle pay \cdot change, \texttt{C} \rangle}), ... \tag{6.10}$$

[2] Remember (from Definition 7) that an FDFA is deterministic when the behaviour of every single product is deterministic. Thus, from one state, we can have multiple transitions with the same action if, and only if, their feature expressions are mutually exclusive.

$Q \leftarrow \{q_p | p \in P\}$;
$q_0 \leftarrow q_{\langle\epsilon,\mathsf{FM}\rangle} \in Q$;
$F \leftarrow \{q_p \in Q | T[p,\epsilon] \neq \texttt{false}\}$;
for $p = \langle seq, \mathsf{fe}\rangle \in P$ **do**
  for $\alpha \in \mathsf{A}$ **do**
    for $p' = \langle seq', \texttt{fe}'\rangle \in \texttt{Targets}(\langle seq, \mathsf{fe}\rangle, \alpha)$ **do**
      $\delta(q_p, \alpha) \leftarrow (\texttt{fe}', q_{p'})$;
    end
  end
end
return $\mathsf{FDFA} = (\mathsf{FM}, \mathsf{A}, Q, q_0, F, \delta)$

Algorithm 3: `MakeConjecture`$(\mathsf{FM}, \mathsf{A}, T, T^+)$

Table 6.7: Building hypothesis from OT

OT		**S**		
		ϵ	$return$	**...**
P	$\langle\epsilon, \mathtt{FM}\rangle$	`false`	`false`	...
	$\langle pay, \mathtt{C}\rangle$	`false`	`false`	...
	$\langle pay \cdot change, \mathtt{C}\rangle$	`false`	`false`	...
	$\langle pay \cdot change \cdot cancel, \mathtt{C}\rangle$	`false`	C	...
	$\langle pay \cdot change \cdot cancel \cdot return, \mathtt{C}\rangle$	C	`false`	...
	...	...	...	...
P$^+$	$\langle pay \cdot change \cdot cancel \cdot return, \neg\mathtt{C}\rangle$	`false`	`false`	...
	...	...	...	...

6.3 Counterexample Analysis and Refinement

We can send the hypothesis FDFA to the Teacher through an **equivalence query** ($\mathcal{EQ}$, Definition 22). The role of $\mathcal{EQ}$ is to compare the hypothesis with the SUL. If the hypothesis matches the system to learn, the execution terminates and returns the FDFA. If not, the teacher sends a counterexample and the learner will pursue the learning. In that case, we extract all `prefixes`of the counterexample sequence (Definition 23). By associating each of them with the feature expression, the algorithm creates new short prefixes for the $\mathcal{OT}$, as shown in Example 6.3.1.

Definition 22 ($\mathcal{EQ}$)**.** The function $\mathcal{EQ} : LFTS \longrightarrow (\mathsf{Seq}(\mathsf{A}^\star) \times \mathbb{B}(\mathsf{F}))$ returns the result of an equivalence query sent by the Learner to the Teacher. Intuitively, the Learner asks if the hypothesis automaton $\mathcal{H}$ is equivalent to the SUL:

$$L(H) = L_{SUL}\,?$$

The Teacher answer either by $(\epsilon, \top)$ if the model is equivalent, or by sending a counterexample $\mathsf{ce} \in \mathsf{Seq}(\mathsf{A}^\star)$ associated with a condition $\mathtt{fe} \in \mathbb{B}(\mathsf{F})$. In the latter case, the counterexample is a sequence of actions that are not accepted by $\mathcal{H}$ whereas accepted by the SUL under the condition `fe`.

Definition 23 (`prefixes`)**.** The function `prefixes` : $\mathsf{Seq}(\mathsf{A}^\star) \longrightarrow \mathsf{Seq}(\mathsf{A}^\star) \times ... \times \mathsf{Seq}(\mathsf{A}^\star)$ returns all the prefixes of a counterexample [3]. For example,

$$\texttt{prefixes}(abcd) = \{a, ab, abc, abcd\}.$$

Example 6.3.1 (Counterexample Analysis)**.** Suppose the equivalence query returns the following counterexample: $\langle pay \cdot change \cdot cancel \cdot return, C\rangle$. We need to add four new prefixes in our table, resulting in Table 6.8. As stated previously, we also need to ensure determinism by adding the prefixes associated with the negation of the feature expression. After this, we can pursue the learning by extending the table and making it closed and consistent again.

Table 6.8: Updating OT with counterexample

OT		**S**	
		ϵ	
P	$\langle \epsilon, \texttt{FM}\rangle$	`false`	
	$\langle pay, \texttt{C}\rangle$	`false`	
	$\langle pay \cdot change, \texttt{C}\rangle$	`false`	
	$\langle pay \cdot change \cdot cancel, \texttt{C}\rangle$	`false`	
	$\langle pay \cdot change \cdot cancel \cdot return, \texttt{C}\rangle$	`C`	
P$^+$	$\langle pay, \neg\texttt{C}\rangle$	`false`	
	$\langle pay \cdot change, \neg\texttt{C}\rangle$	`false`	
	$\langle pay \cdot change \cdot cancel, \neg\texttt{C}\rangle$	`false`	
	$\langle pay \cdot change \cdot cancel \cdot return, \neg\texttt{C}\rangle$	`false`	
	...	...	

Analysing counterexamples directly impacts the Observation Table (*i.e.,* by adding new prefixes) and thus, leads to the construction of a new hypothesis proposal.

6.4 Complete Algorithm

In the previous sections, we have described in detail all the different parts forming our new algorithm FL^*. Once we put those parts together, we obtain Algorithm 4.

In the produced automaton, each transition guard is a feature expression only composed of conjunctions ($\wedge$). It means that from one state, we can have multiple transitions with the same action but with distinct and disjoint feature expressions. This ensures determinism throughout the learning. However, once the algorithm completes, the resulting automaton tends to be unnecessarily large. To address this, we have defined a procedure to reduce its size, by merging states targeted by a set of such transitions. Currently, our procedure allow to add disjunctions ($\vee$). In the

[3] We omit the empty sequence ϵ which is already in the table, from the initialisation step. Moreover, the empty prefix is always $\langle \epsilon, \texttt{FM}\rangle$ since it defines the initial state, which is part of any behaviour respecting the FM.

$P \leftarrow \{\langle \epsilon, \texttt{FM}\rangle\}$; $S \leftarrow \{\epsilon\}$; $P^+ \leftarrow \{\}$; ▷ $\mathcal{OT}$ initialisation
$T[\langle \epsilon, \texttt{FM}\rangle, \epsilon] \leftarrow \mathcal{MQ}(\langle \epsilon, \texttt{FM}\rangle, \epsilon)$; ▷ $\mathcal{OT}$ initialisation
while $\exists$ (ce, fe) **do**
  while !closed $\vee$!consistent **do**
    $P^+ \leftarrow P^+ \cup P \cdot \mathsf{A}$; ▷ $\mathcal{OT}$ Extension
    $\forall p \in P, \forall s \in S,\ T[p,s] \leftarrow \mathcal{MQ}(p,s)$; ▷ $\mathcal{OT}$ Extension
    $\forall p^+ \in P^+, \forall s \in S,\ T^+[p^+,s] \leftarrow \mathcal{MQ}(p^+,s)$; ▷ $\mathcal{OT}$ Extension
    if $\exists p^+ \in P^+$ **such that** $\forall p \in P, row(p^+) \neq row(p)$ **then** ▷ Making $\mathcal{OT}$ closed
      $P \leftarrow P \cup \{p^+\}$; ▷ Making $\mathcal{OT}$ closed
      $P^+ \leftarrow P^+ \setminus \{p^+\}$; ▷ Making $\mathcal{OT}$ closed
    else ▷ Making $\mathcal{OT}$ closed
      Mark as closed; ▷ Making $\mathcal{OT}$ closed
    end ▷ Making $\mathcal{OT}$ closed
    if $\exists p_1, p_2 \in P \cup P^+$ **such that:** ▷ Making $\mathcal{OT}$ consistent
    $p_1 = \langle seq_1, \texttt{fe}_1\rangle,\ p_2 = \langle seq_2, \texttt{fe}_2\rangle,\ \big(\texttt{row}(p_1) = \texttt{row}(p_2)\big)$ ▷ Making $\mathcal{OT}$ consistent
    $\wedge \big(\exists s \in S, \exists \alpha \in A, \bigvee_{p \in \mathcal{C}_1} T[p,s] \neq \bigvee_{p \in \mathcal{C}_2} T[p,s]\big)$ ▷ Making $\mathcal{OT}$ consistent
    with ▷ Making $\mathcal{OT}$ consistent
    $\mathcal{C}_1 = \big\{p \in P \cup P^+ | p = \langle seq_1 \cdot \alpha, \texttt{fe}\rangle,\ \texttt{fe} \in \mathbb{B}(\mathsf{F})\big\}$ ▷ Making $\mathcal{OT}$ consistent
    $\mathcal{C}_2 = \big\{p \in P \cup P^+ | p = \langle seq_2 \cdot \alpha, \texttt{fe}\rangle,\ \texttt{fe} \in \mathbb{B}(\mathsf{F})\big\}$ ▷ Making $\mathcal{OT}$ consistent
    then ▷ Making $\mathcal{OT}$ consistent
      $S \leftarrow S \cup \{\alpha \cdot s\}$; ▷ Making $\mathcal{OT}$ consistent
      $\forall p \in P,\ T[p, \alpha \cdot s] \leftarrow \mathcal{MQ}(p, \alpha \cdot s)$; ▷ Making $\mathcal{OT}$ consistent
      $\forall p^+ \in P^+,\ T^+[p^+, \alpha \cdot s] \leftarrow \mathcal{MQ}(p^+, \alpha \cdot s)$; ▷ Making $\mathcal{OT}$ consistent
    else ▷ Making $\mathcal{OT}$ consistent
      Mark as consistent; ▷ Making $\mathcal{OT}$ consistent
    end ▷ Making $\mathcal{OT}$ consistent
  end
  $Q \leftarrow \{q_p | p \in P\}$; ▷ Building Hypothesis automaton
  $q_0 \leftarrow q_{\langle \epsilon, \texttt{FM}\rangle} \in Q$; ▷ Building Hypothesis automaton
  $F \leftarrow \{q_p \in Q | T[p,\epsilon] \neq \texttt{false}\}$; ▷ Building Hypothesis automaton
  for $p = \langle seq, \texttt{fe}\rangle \in P$ **do** ▷ Building Hypothesis automaton
    for $\alpha \in \mathsf{A}$ **do** ▷ Building Hypothesis automaton
      for $p' = \langle seq', \texttt{fe}'\rangle \in \texttt{Targets}(\langle seq, \texttt{fe}\rangle, \alpha)$ **do** ▷ Building Hypothesis automaton
        $\delta(q_p, \alpha) \leftarrow (\texttt{fe}', q_{p'})$; ▷ Building Hypothesis automaton
      end ▷ Building Hypothesis automaton
    end ▷ Building Hypothesis automaton
  end ▷ Building Hypothesis automaton
  $\mathcal{H} \leftarrow \mathsf{FDFA}(\mathsf{FM}, \mathsf{A}, Q, q_0, F, \delta)$; ▷ Building Hypothesis automaton
  $(\texttt{ce}, \texttt{fe}) \leftarrow \mathcal{EQ}(\mathcal{H})$; ▷ $\mathcal{EQ}$ and counterexample analysis
  $P \leftarrow P \cup \{\langle p, \texttt{fe}\rangle | p \in \texttt{prefixes(ce)}\}$; ▷ $\mathcal{EQ}$ and counterexample analysis
end
return $\mathcal{H}$

Algorithm 4: Detailed pseudo-code of FL^* algorithm

future, one could imagine a more elaborate simplification mechanisms, allowing higher level connector (*e.g.,* $\oplus$, the XOR operator).

Rather than travelling through the automaton itself, we directly modify the observation table. This approach eliminates the need for an expensive merging automaton algorithm. Additionally, since the table already encompasses all the necessary information in a more compact data structure, it is easier to manipulate.

In Algorithm 5, we create a new observation table by merging closed prefixes (*i.e.,* future states). First, we select all the prefixes with the same associated sequence (in Algorithm 5, this set is noted $\mathcal{C}$). Then, we take the disjunction of all the associated feature expressions $\left(\bigvee_{\langle seq_{simp},\texttt{fe}\rangle\in\mathcal{C}}\texttt{fe}\right)$. The initial sequence and this new feature expression form the new prefix, merged from the set $\mathcal{C}$ of prefixes. A similar treatment is applied on the table cells to fill the new row: for each suffix, we take the disjunction of all corresponding cells $\left(\bigvee_{p\in\mathcal{C}} T[p,s]\right)$. For extended prefixes, we only copy rows defining a transition (*i.e.,* we suppress entire lines of `false` values). This new table is now complete and we can apply Algorithm 3 as previously to transform the table into an automaton. An example of the result of this procedure is presented in Appendix B.

$S_{simp} \leftarrow S$;
$P_{simp} \leftarrow \left\{p_{simp} = \langle seq_{simp}, \texttt{fe}_{simp}\rangle | \texttt{fe}_{simp} = \bigvee_{\langle seq_{simp},\texttt{fe}\rangle\in\mathcal{C}}\texttt{fe}\right\}$ with $\mathcal{C} = \left\{\langle seq_{simp}, \texttt{fe}\rangle \in P | \texttt{fe} \in \mathbb{B}(\mathsf{F})\right\}$
for $p_{simp} = \langle seq_{simp}, \texttt{fe}_{simp}\rangle \in P_{simp}$ **do**
  for $s \in S_{simp}$ **do**
    $T^+_{simp}[p_{simp}, s] \leftarrow \bigvee_{p\in\mathcal{C}} T[p,s]$ with $\mathcal{C} = \left\{\langle seq_{simp}, \texttt{fe}\rangle \in P, \texttt{fe} \in \mathbb{B}(\mathsf{F})\right\}$;
  end
end
$P^+_{simp} \leftarrow \left\{p^+ \in P^+ | \exists s \in S,\ T^+[p^+, s] \neq \texttt{false}\right\}$;
for $p^+ \in P^+_{simp}$ **do**
  for $s \in S_{simp}$ **do**
    $T^+_{simp}[p^+, s] \leftarrow T^+[p^+, s]$;
  end
end
return $\mathcal{OT}_{simp} = (T_{simp}, T^+_{simp})$

Algorithm 5: `Simplify`$(\mathcal{OT}(T, T^+))$

6.5 Adaptive Learning versus Variability-Aware Learning

In this section, we present the related work of Tavassoli, Damasceno, *et al.*, which is closely aligned with our research. We discuss their approaches in chronological order, with each section focusing on one of their contributions [54, 56, 201]. We wrap-up this section by discussing points of comparison with our new algorithm, FL^*.

6.5.1 Learning to Reuse

Adaptive learning is a variant of model learning that aims to accelerate the learning process by leveraging pre-existing models from previous or alternative versions, rather than starting from scratch. Adaptive learning is commonly employed in scenarios characterised by time-variability, where the goal is to compare different versions of the same system captured at different timestamps. In 2019, Damasceno *et al.* [54] propose a new algorithm for adaptive learning called partial-Dynamic L^*_M (∂L^*_M). The novelty of this approach lies in the reusability of the learned components. By leveraging the adaptability of the models and their shared components, it becomes possible to reuse and transfer knowledge across different product variant. This reusability aspect adds significant value to the overall approach, as it promotes efficiency and reduces redundancy in the learning process. As a matter of course, ∂L^*_M is well suited for individually learning SPL products (product-based approach).

The key feature of ∂L^*_M is its ability to explore observation tables on-the-fly to eliminate redundant prefixes and deprecated suffixes. This approach significantly reduces the number of membership queries required for learning a new variant. Additionally, by bypassing the initial rounds of learning, it reduces the number of equivalence queries needed. Through the efficient pruning of unnecessary queries, ∂L^*_M accelerates the learning process and improves overall efficiency.

State-of-the-art adaptive learning algorithms often struggle with software evolutions. As the states of a SUL is altered, the number of suffixes with good quality decreases. Therefore, when older versions are reused, a higher number of irrelevant queries is expected. However, an evaluation on 18 distinct FSMs representing different versions of the OpenSSL server-side demonstrated that ∂L^*_M exhibited less sensitivity to software evolution. Moreover, it showcased greater efficiency and required fewer membership queries compared to other state-of-the-art adaptive learning algorithms.

6.5.2 Learning by Sampling

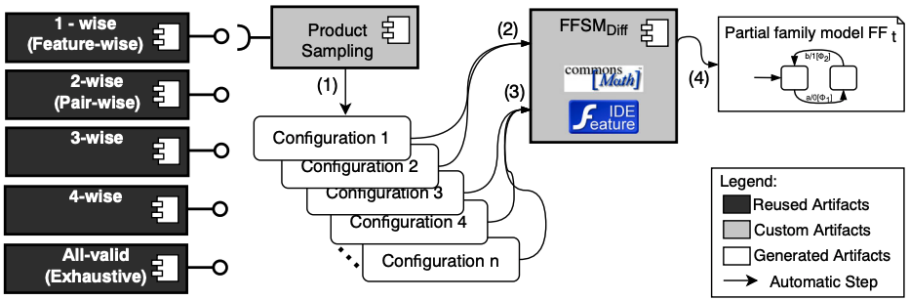

Figure 6.2: Adaptive learning of SPL: the $FFSM_{Diff}$ framework [56]

To the best of our knowledge, the first attempt to learn a family-based behavioural model for a VIS was born in 2021. Damasceno *et al.* [56] presented $FFSM_{Dif}$ which aims to abstract an FFSM from individually learned or hand-crafted

product models to learn the behaviour of succinct family model. They rely on L^* to learn individual product models (*a.k.a.* FSM), that are progressively merged to the family model (*a.k.a.* FFSM). Their process relies on the following steps, summarised in Figure 6.2:

(i) Using the FM, sample SPL products for a maximum feature coverage;
(ii) Apply L^* on product p_1 and p_2 to learn a first FFSM;
(iii) Iteratively include behaviour of products $p_3, ...p_n$, into the existing FFSM.

The $FFSM_{Dif}$ algorithm is implemented on top of the LearnLib framework [179]. Damasceno *et al.* also analyse the effect of T-wise sampling [126, 176] to efficiently select individual products, comparing results for $T \in \{1,2,3,4\}$. Figure 6.3 presents the number of variants included in the learning process, for each evaluated case study and sampling criterion.

SPL	Size of the sampled subset generated by T-wise				
	Feature-wise	Pair-wise	3-wise	4-wise	All-valid
AGM	3	6	6	6	6
VM	2	6	13	19	20
WS	2	5	8	8	8
AEROUC5	3	6	9	9	9
CPTERMINAL	3	8	16	24	30
MINEPUMP	3	7	13	24	32

Figure 6.3: Number of sample variant generated by each sampling criterion [56]

While sampling techniques are commonly used when the budget is limited to the coverage of a few variants, it is important to emphasise that these techniques solely rely on structural variability (*i.e.,* the FM). While sampling ensures dissimilarity among features, it does not guarantee dissimilarity at the behavioural level [67]. Moreover, these techniques operate under the assumption that exhaustive learning may be feasible for small product lines but becomes impractical for larger ones. This hypothesis may hold true for $FFSM_{Dif}$ since, despite learning a family model, it fundamentally follows a product-based approach. In other words, products are learned individually and subsequently merged. However, this approach may appear to contradict the core essence of the SPL paradigm, where the family is the central aspect and should be treated as such. In SPL, the emphasis is on capturing the commonalities and variabilities across the entire product line, rather than treating each product as an isolated entity. Therefore, an approach that treats products individually and later merges them may not fully align with the holistic nature of SPL.

6.5.3 Adaptive Behavioural Model Learning

Recently, Tavassoli *et al.* [201] introduced the PL^* algorithm. PL^* novelty lies in its ability to store previous observation tables into a repository. Figure 6.4 represents

the following process:

Initialisation for product p_1:

(i) Apply learn p_1 FSM (M_1). For this step, initialising the initial set of suffixes with the alphabet of the product is the only optimisation from the original L^*.

(ii) Initialise the repository by adding the learned table: $\mathcal{OT}_{Repository} = \{\mathcal{OT}_1\}$.

Loop and apply PL^* over product p_i, $\forall i \in \{2, ..., n\}$:

(iii) Initialise $\mathcal{OT}_i$ using $\mathcal{OT}_{Repository} = \{\mathcal{OT}_1, ..., \mathcal{OT}_{i-1}\}$:

  (a) Defines the set of prefixes $S_i = \bigcup_{j \in \{1,...,i-1\}} S_j$, restricted to prefixes that solely comprises input symbols from the alphabet of p_i;

  (b) Initialise the set of suffixes E_i with the alphabet of p_i;

  (c) Add $\bigcup_{j \in \{1,...,i-1\}} E_j$, restricted to suffixes that solely comprises input symbols from the alphabet of p_i, to the set of suffixes E_i.

(iv) Obtain M_i, the FSM of (p_i).

(v) Add $\mathcal{OT}_i$ to the repository: $\mathcal{OT}_{Repository} = \{\mathcal{OT}_1, ..., \mathcal{OT}_i\}$.

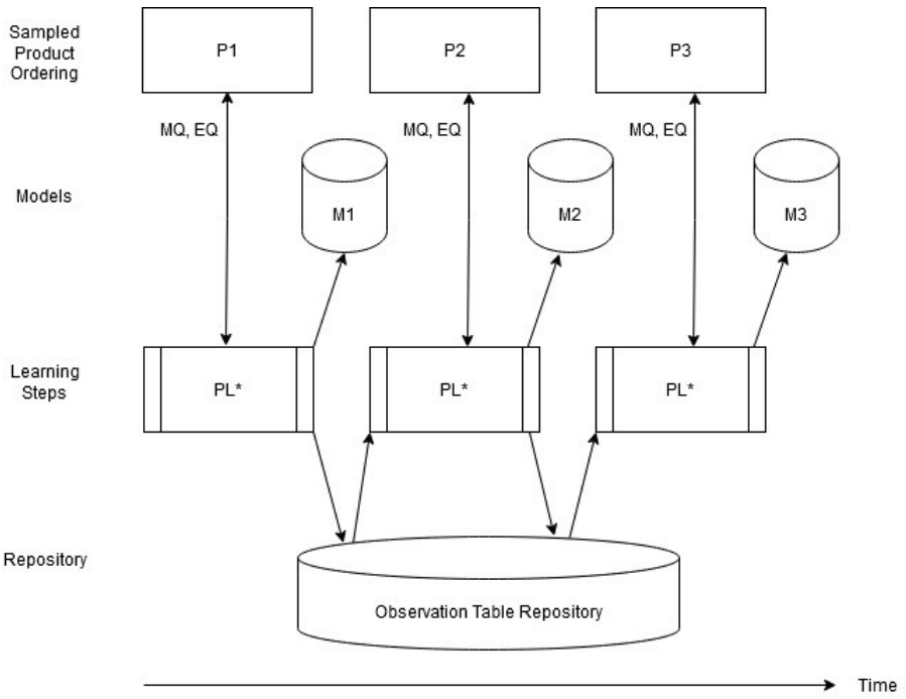

Figure 6.4: Adaptive learning of SPL: the PL^* framework [201]

Compared to $FFSM_{Dif}$, PL^* leverages the commonalities between variants at an earlier stage of the learning process. It considers products individually and

incorporates the reuse of previous artifacts when learning subsequent products. However, it is important to acknowledge that this approach still maintains a product-oriented perspective. Furthermore, while $FFSM_{Dif}$ generates a single FFSM as its output, PL^* produces a collection of FSMs. In this sense, PL^* can be seen as an improvement over ∂L^*_M rather than $FFSM_{Dif}$. It aims to enhance the capabilities of ∂L^*_M by taking advantage of the commonalities between variants earlier in the process, while still preserving the individual FSM representation for each product.

6.5.4 Comparison with FL^*

Although Tavassoli, Damasceno, *et al.* [54, 56, 201] are interested in learning behavioural model(s) of an SPL, their methods differ from ours in several ways.

Different output models. While our focus has been on FTSs (and FDFAs), the existing work primarily revolves around (F)FSMs. As discussed in Section 1.3, (F)FSMs and (F)TSs exhibit different expressive capabilities, which prevents a thorough comparison between FL^* and the different algorithms presented in Sections 6.5.1 to 6.5.3. In FL^*, we adopt a different approach by not considering separate input and output alphabets. Instead, we employ a unified alphabet of actions that the system can either accept or reject. To put it simply, although we have a complete input alphabet, the output alphabet is limited to two values: “accept” or “reject”.

Model collection versus unified model. Moreover, in two out of the three approaches, the objective slightly differs. Instead of learning a single model, where behaviour is linked to structural variability, ∂L^*_M [54] and PL^* [201] learn a collection of individual FSMs. Although it is still possible to merge them subsequently, this process requires an additional and relatively expensive step.

Product versus family. Finally, there is a fundamental difference between FL^* and these works. As we discussed in Section 6.5.2, those three approaches are in fact product-based approach that leverage commonalities. In contrast, FL^* aims to present a more comprehensive and integrated approach, where feature expression are treated as first-class citizens. The notion of software family becomes the core of our framework, emphasising the importance of considering the relationships between features.

CHAPTER 7

IMPLEMENTATION AND CASE STUDIES

This chapter presents a new tool, called LiFTS, implementing FL^*. We present its architecture and how to use it. Then, we describe the different case studies used to assess LiFTS. We consider models from different sources with varying characteristics. These systems were mostly described in previous work on FTS [44, 61].

7.1 LiFTS

LiFTS is a Java implementation of the FL^* algorithm. It is designed as a Maven project and reuses the VIBeS framework [62, 66], for specific FM, FTS and feature expressions utilities. To get a unique representation of FMs and feature expressions, we represent them as **Binary Decision Diagrams (BDDs)**, implemented by the JavaBDD library [1]. Uniqueness is the key to compare feature expressions and check for equality.

7.1.1 Architecture

Figure 7.1 presents the general architecture of LiFTS, composed of two root classes (`Main` and `Learner`) and three primary packages:

- `data` package: contains the data structures used in FL^*, such as `ObservationTable`, `Prefix`, `Suffix`, and others.
- `io` package: contains the SUL interface, but also utilities to interact with input and output files, in XML or CSV format;
- `vibes_mock` package: contains extensions and modifications of the VIBeS library.

[1] https://github.com/com-github-javabdd/com.github.javabdd

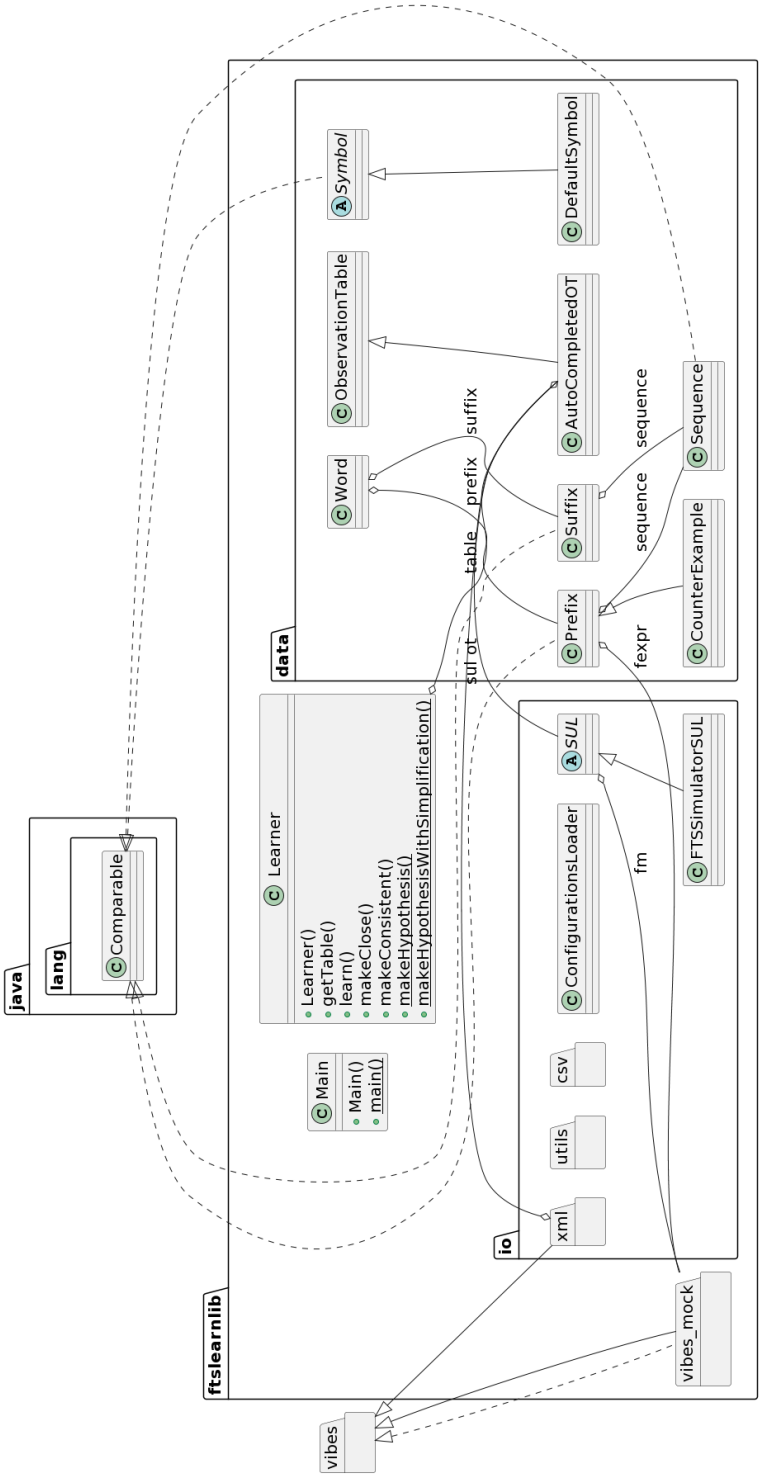

Figure 7.1: Class Diagram

During execution, the `main` function (in the Main class) instantiates a Learner which maintains an Observation Table. The `learn` function implements the algorithm presented in Chapter 6, making calls to `makeClose`, `makeConsistent` and `makeHypothesis` functions. Once it terminates, it performs `makeHypothesisWithSimplification` to obtain the final FDFA.

In the `data` package, the `ObservationTable` stores Prefixes and Suffixes in a Guava Table. To enhance its functionality, we have introduced the `AutoCompletedOT` class, which extends this structure to support automatic membership queries. This enables the table to be automatically completed with feature expressions whenever a prefix or suffix is added, ensuring that both rows (in the case of prefixes) and columns (in the case of suffixes) are fully populated with the necessary information.

In the `io` package, CSV utilities allow to store and load `ObservationTable`, while XML utilities interact with the FM and the FTS. The abstract class `SUL` facilitates generalisation by separating the rest of the implementation from the specific implementation of the system or from the system simulator. In L^* metaphor, this class plays the role of the Teacher. The `SUL` provide interfaces for memberships queries and equivalence queries. Currently, there is only one concrete implementation of `SUL`, which simulates the system based on an FTS.

Example generation The `FTSSimulatorSUL` is an implementation of the SUL based on a reference FTS [2]. During its initialisation, it generates random examples (*i.e.*, execution traces) which are used for equivalence query (*i.e.*, they are potential counterexamples). The `FTSSimulatorSUL` requires an FM, an FTS and a number of positive (*i.e.*, accepted by the SUL) and negative examples (*i.e.*, rejected by the SUL). Figure 7.2 presents a generalisation of this counterexample generation, apply to the soda vending machine example.

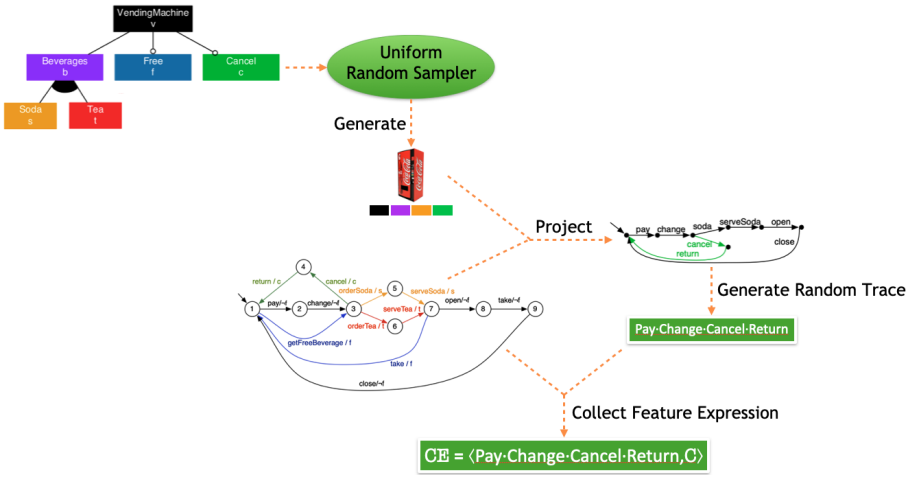

Figure 7.2: Generalisation of the counterexample generation process

[2] This hypothesis is non-trivial and its implications will be further discussed in Section 9.3.

To generate positive examples, the simulator execute the following steps:

(i) Generate a product configuration from the FM, using a uniform sampler (CMSGen [3]).
(ii) Derive the product TS through projection;
(iii) Start from the initial state of the TS;
(iv) Randomly choose an outgoing transition, until either returning to the initial state (as TSs are infinite, Devroey [61] indicates that the last transition of a valid trace must always end up in the initial state) or reaching a trace length threshold.

The trace length threshold serves as a safeguard against infinite loops, where the system continuously takes the same branch within a loop. When the threshold is reached, we initiate a search for a new trace starting from the same initial product. On the other hand, if the threshold is not reached, we consider the generated positive example as valid and return it for further use. This mechanism ensures that we can generate diverse and representative positive examples while avoiding scenarios where the system gets trapped in endless loops. Note that a reset is required each time we reach the threshold or for each new counterexample.

To generate negative examples, the procedure follows a similar approach:

(i) Generate a product configuration from the FM, using a uniform sampler (CMSGen [4]);
(ii) Derive the product TS through projection;
(iii) Randomly select a number n of actions within the maximum trace length threshold;
(iv) Start from the initial state of the TS;
(v) Randomly choose an outgoing transition, until either returning to the initial state or reaching n actions.

To ensure that the negative examples exhibit invalid behaviour, we impose that the generated example is only considered if the final state (after performing n actions) differs from the initial state. If the final state matches the initial state, we return to step iii to generate a new negative example. This iterative process guarantees that the negative examples demonstrate the absence or deviation from the expected behaviour, providing valuable counterexamples that contribute to the learning algorithm's understanding of the SUL's limitations and failure cases. Here again, it requires frequent resets of the system.

Memberships query A `Word` is composed of a pair of `Prefix` and `Suffix`. The `FTSSimulatorSUL` concatenates the action sequences from both of them and tries to execute the resulting sequence on the FTS. During this process, it collects the feature expression of the generated trace. If the sequence is not accepted or does not end in the initial state (indicating an incomplete word), the membership query

[3] https://github.com/meelgroup/cmsgen
[4] https://github.com/meelgroup/cmsgen

returns `false`. Otherwise, it returns the conjunction between the collected feature expression and the feature expression of the prefix.

Equivalence query The `CounterExample` class already contains the result for the SUL indicating whether the example is accepted or refused by the system. Consequently, the equivalence query function aims to execute the example on the hypothesis and check whether the results match. If the results are divergent, indicating a difference between the hypothesis and the SUL, the example is returned as an actual counterexample. Alternatively, if the results match, the example is ignored, and the next example is considered. The execution stops when all the examples have been considered.

Feature expression In the `vibes_mock` package, we have introduced a new class to handle feature expressions that better aligns with our requirements. The existing `FExpression` class in VIBeS was not suitable for our needs due to the following reasons:

(i) The original `FExpression` class allowed the definition of feature expressions that did not adhere to the FM, requiring manual checks for FM satisfiability whenever the expressions were used.
(ii) Additionally, the `FExpression` class relied on the Jbool Expressions library [5], which determined equality based solely on syntax. This meant that feature expressions constructed in different ways, such as having operators added in a different order, were considered different even when they represented the same logical expression.

The issue mentioned in point ii was particularly problematic because it could lead to the addition of prefixes that already existed in the table but in a different form. This violated determinism and caused inconsistencies. To address this, we developed a new class that constructs feature expressions directly from the FM, using a Binary Decision Diagrams (BDD) internal representation [6].

Using a BDD structure allows better comparison of feature expressions. Each feature expression is assigned a unique representation based on its semantics, which corresponds to the set of products it represents. Importantly, this uniqueness is maintained **for a given feature order**. We currently assume that the feature order is provided by domain experts, whose mental representation of the FM is a tree-like structure. Our heuristic is based on the assumption that the experts have selected a convenient feature order that leads to interesting simplifications when branches are cut in the BDD representation.

7.1.2 Usage

LiFTS can be executed from the command line. After downloading the sources, we can build with Maven:

[5] https://github.com/bpodgursky/jbool_expressions
[6] https://github.com/com-github-javabdd/com.github.javabdd

```
mvn clean validate compile test package
```

If we do not want to execute the test, we can add the `-DskipTests` option. Once the build is completed, LiFTS can be launched and parameterised as described by Table 7.1.

Table 7.1: Usage of LiFTS.

```
usage:  java -jar
target/LiFTS-1.0-SNAPSHOT-jar-with-dependencies.jar -help |
-d | -sul (-ex)?  (-o)?
```

`-d`	Specify we want to use all predefined models with default number of counterexamples and output directory.
`-ex <nbPosExamples> <nbNegExamples>`	Specify the number of traces used to generate positive and negative counterexamples. Default values are 200 for both positive and negative counterexamples.
`-help`	Prints this help message.
`-o <outputDir>`	Specify the output directory. If this option is not provided, *target/output/* is used as default.
`-sul <sulName>`	Specify the name of the SUL. This name should correspond to two files: • `sulName.fts`: an XML file storing an FTS used as a simulator and which is placed under *src/main/resources/fts/*; • `sulName.dimacs`: a DIMACS file storing the FM of the system and which is placed under *src/main/resources/fm/*.

7.1.3 Automaton Visualisation

XML files are convenient to generate and to use as software input, but they can be challenging to handle when it comes to manual comparison. As our models quickly grow in size, visualisation becomes a concern. We require a tool that can read XML input files and convert their content into an automaton with states (without specifying exact coordinates for each node) and labelled transitions, while also displaying feature expressions. We chose the platform Neo4J [7], which fulfills (almost) all our requirements. Neo4J is a a graph database management system, implemented in Java, which allows us to interact with it using the Cypher query language [8]. It is widely used by many prominent companies worldwide.

[7] https://neo4j.com/product/neo4j-graph-database/
[8] https://neo4j.com/product/cypher-graph-query-language/

```
1  WITH "filePath.fts" AS uri
2  CALL apoc.load.xml(uri) YIELD value
3  UNWIND value._children AS fts
4  WITH fts
5  UNWIND fts._children AS states
6  CALL apoc.create.node(['State',states._id], {name:states.id}) YIELD
       node
7  MERGE (s:State {name: node.name})
8  WITH s, states._children AS tr_list
9  UNWIND tr_list AS transition
10 WITH s, transition
11 MERGE (st:State {name: transition.target})
12 WITH s, st, transition
13 CALL apoc.merge.relationship(s, transition.action, {fexpression:
       transition.fexpression}, {}, st) YIELD rel
14 RETURN s, st, rel
```

Listing 7.1: Load an FTS in a Neo4J Database with Cypher

We have defined a Cypher procedure (Listing 7.1) to load an FTS (or FDFA) into a graph database. This procedure represents states as nodes, transitions as relationships and feature expressions as properties of these relations. We can now visualise and interact with our automaton, such as moving nodes, renaming or removing elements, changing colours and styles, and more. This capability provides a clear understanding of the model and facilitates comparison between different versions. One minor issue is that relation properties cannot be directly displayed on the edges themselves. To access the feature expression associated with a transition, we first need to select the transition (see an example in Figure 7.3). In practice, this limitation does not significantly affect our work. However, it is worth noting that in this thesis, we will settle down for the structure of the automaton, without including annotations of feature expressions. Readers who desire the complete FDFA are encouraged to refer to the supplementary materials.

7.2 Case Studies

7.2.1 Forum

The first example was created specifically for the purpose of LiFTS and helped in the definition of our algorithm. We required a system with few features, but also few actions (in order to limit the size of the input alphabet). This ensures that the learning converges within a few rounds. We thus defined an FM (Figure 7.4) composed of 5 features (including 2 abstract ones). A *Forum* is a place for *Anonymous* or *RegisteredUser* to exchange messages. The system may have a *Cancellation* functionality to suppress messages that have not yet been submitted on the platform.

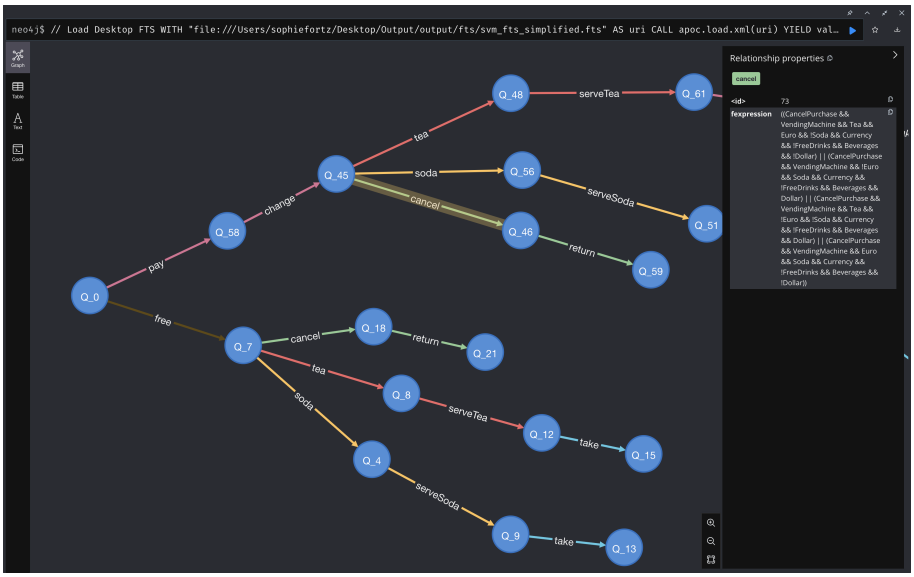

Figure 7.3: Select a transition to get the associated feature expression

Figure 7.5 shows the automata corresponding to the *Forum* behaviour. Before sending a message, users should either log in to the platform through their credentials or choose a temporary pseudo. After they typed their message, they can either cancel (if the *Forum* allows it) or send their message.

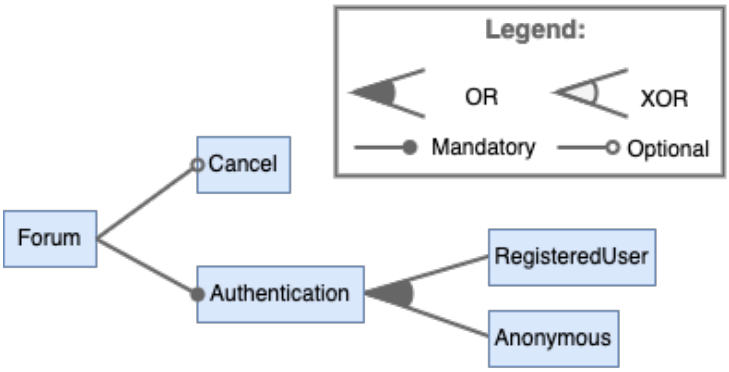

Figure 7.4: Forum FM

7.2.2 Soda Vending Machine

The Soda Vending Machine SPL (SVM), as described by Classen *et al.* [44, 61], is a traditional vending machine for beverages. The corresponding FM can be seen in Figure 7.6. It offers soda and tea, in either euro or dollar currency. Additionally, it

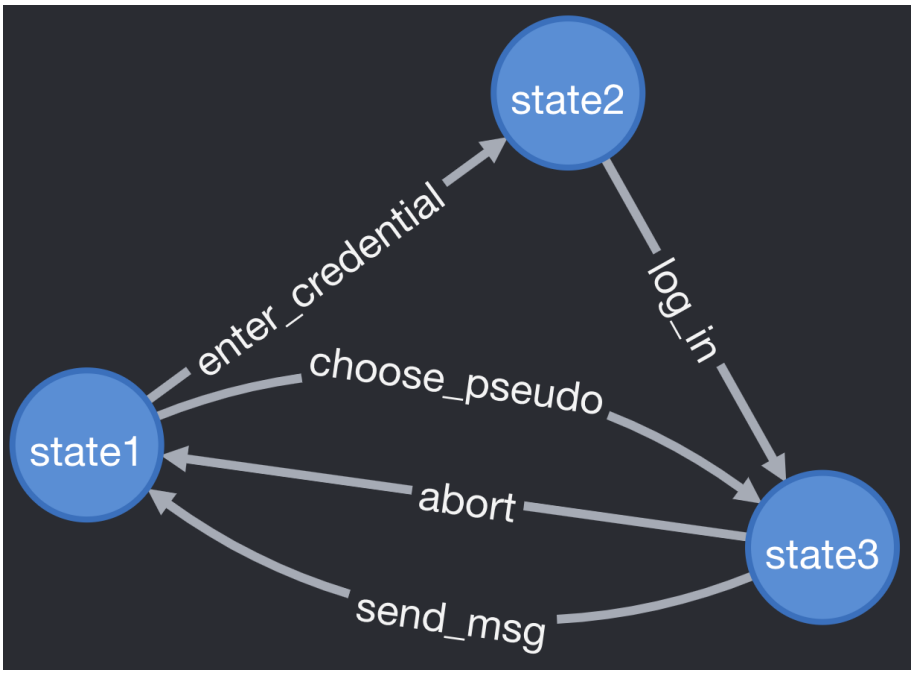

Figure 7.5: Forum FTS

may provide free drinks or may allow users to cancel an ongoing purchase. The behaviour of the SPL is captured by the FTS shown in Figure 7.7.

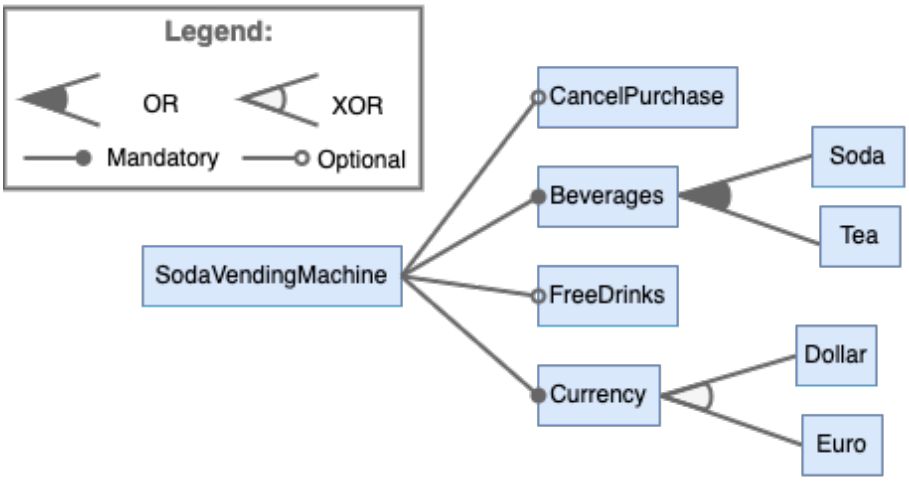

Figure 7.6: Soda Vending Machine FM [44, 61]

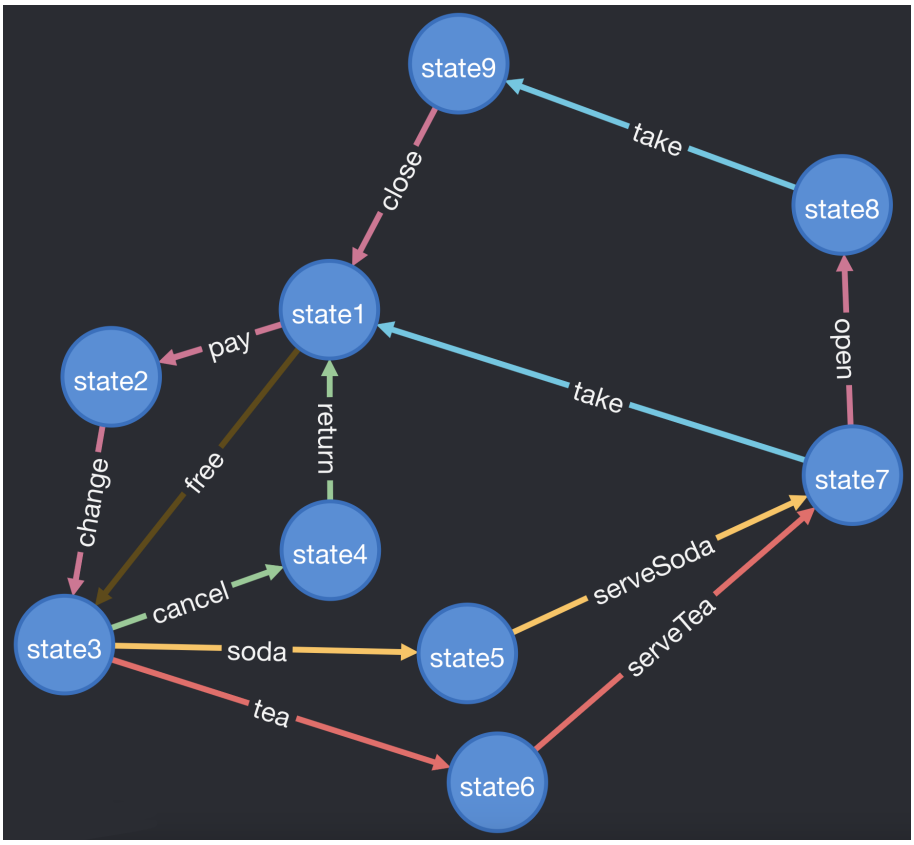

Figure 7.7: Soda Vending Machine FTS [44, 61]

7.2.3 Minepump

The Minepump model [44, 61] is a product line of pumps designed for keeping mine shafts clear of water and preventing the danger of a methane explosion. Figure 7.8 displays the FM of this SPL. Each pump in the system is equipped with a water regulator that can detect the water level in the shaft. Additionally, it may include a methane concentration sensor and a command interface that enables manual control of the pump's start and stop functions. The system should activate the pump once the water level surpasses a predetermined threshold, but only if the methane concentration remains below a critical limit. The minepump operates in a distributed manner, where the controller and sensors function as individual subsystems that communicate through message passing. The behaviour of the pumps is described by the FTS depicted in Figure 7.9.

7.2.4 Card Payment Terminal

The Card Payment Terminal (CP Terminal) [61] describes a card payment machine based on the Eurocard-Mastercard-Visa system. Payments can be done through

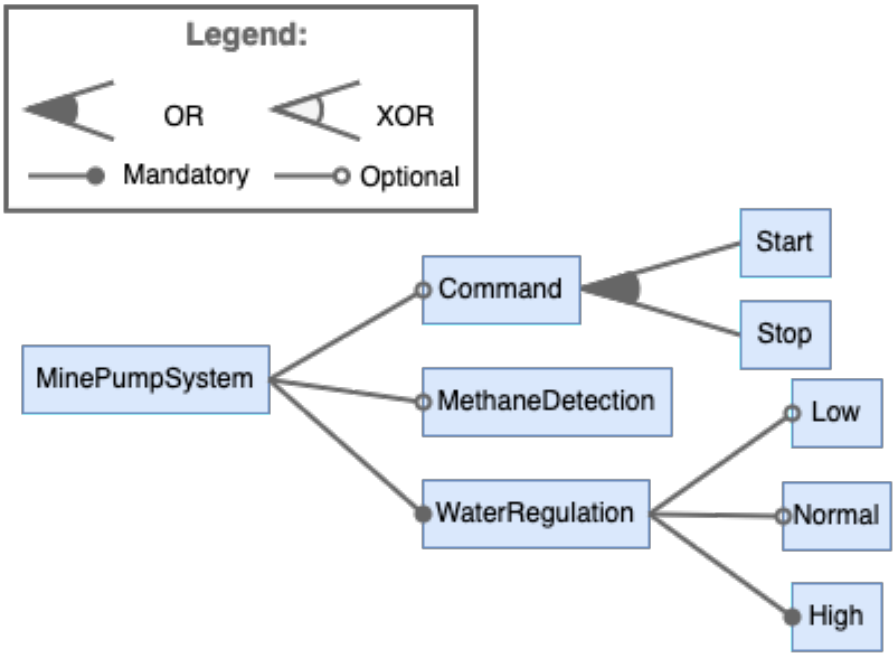

Figure 7.8: Minepump FM [44, 61]

debit or credit. The terminal can access the Internet using different connectivity options or work in an offline mode. It can read cards with their chip, their magnetic strip or through near field contact (NFC). Users should authenticate using their signature or possibly their PIN code. The Card Payment Terminal FM is presented in Figure 7.10 and its behaviour is described by the FTS shown in Figure 7.11.

7.2.5 Sferion™ Landing Symbology Function

Sferion™ is a project of a pilot assistance system from Airbus Defence and Space. Sferion™aims to protect helicopters operating in restricted visibility conditions, such as fog, snowstorm or rain. During the landing, the landing symbology function marks the intended landing position on the ground to help the pilot. This functionality uses a head-tracked Helmet Mounted Sight and Display (HMS/D) and Hands-On Collective And Stick (HOCAS). *SI_sensor_based* or *SI_from_DB* can provide slope indication for landing position. An Obstacle Warning System (OWS) uses sensors to detect the ground and other obstacles in the landing area: either *ELOP* or *HELLAS* sensors. The helmet can also provide visual 3D cues with optionally real reference to the object to enhance spatial awareness. As an additional safety measure, the co-pilot of the helicopter can also validate the marked position for landing. An FM (Figure 7.12) and an FTS (Figure 7.13) was proposed by Devroey [61] to model the Sferion™ landing symbology function.

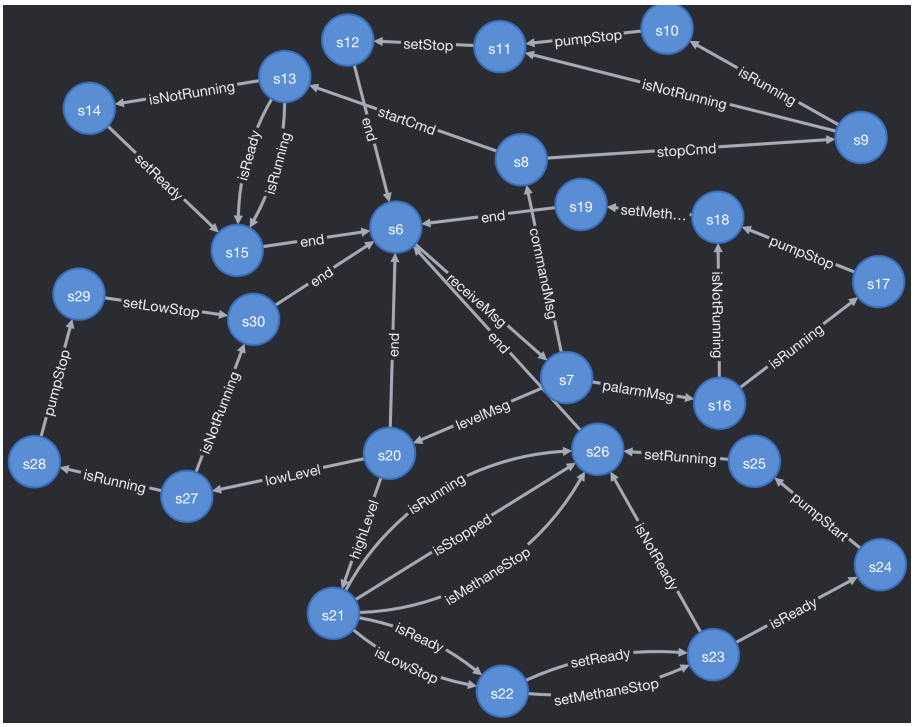

Figure 7.9: Minepump FTS [44, 61]

Table 7.2: Characteristics of the FM for each case study

Model	Features	Mandatory	Optional	Constraints	Products
Forum	5	1	1	0	6
SVM	9	2	2	0	24
Minepump	9	2	4	0	32
CP Terminal	21	4	2	4	4,774
Sferion™	25	10	2	7	64

7.2.6 Models Characteristics

To conclude this chapter, Table 7.2 presents the main characteristics of the different FMs we just described. For each FM, we give the number of features (**Features**), the number of mandatory (**Mandatory**) and optional (**Optional**) features in the model, the number of additional constraints (*i.e.,* that are not displayed in the feature diagram, **Constraints**) and the number of valid products (**Products**) defined by this FM. Except for the Forum case study, all metrics were computed by Devroey [61].

Similarly, we presented the characteristics of each FTS in Table 7.3. We detail the number of states (**States**); the number of transitions (**Transitions**); the number of actions (**Actions**) for each model. The average degree (**Avg. degree**) corresponds to the average number of incoming/outgoing transitions per state (*i.e.,* the number of transitions divided by the number of states). **BFS height** is the maximal number

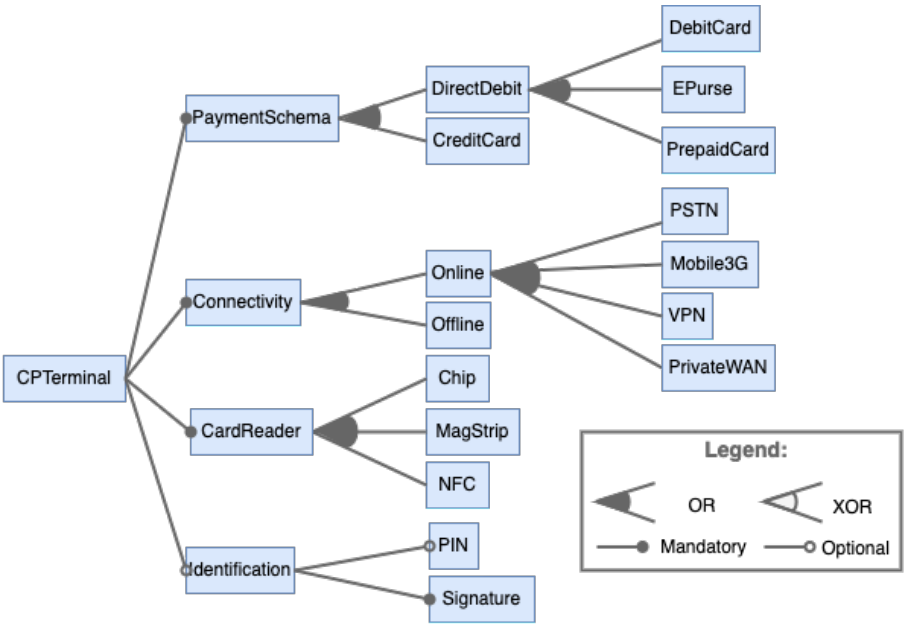

Figure 7.10: Card Payment Terminal FM [61]

Table 7.3: Characteristics of the original FTS for each case study

Model	States	Transitions	Actions	Avg. degree	BFS height
Forum	5	5	5	1	2
SVM	9	13	13	1.44	5
Minepump	25	41	24	1.64	15
CP Terminal	11	17	16	1.54	7
Sferion™	25	46	12	1.84	16

of states between the initial state and another state when traversing the FTS in a breadth-first manner.

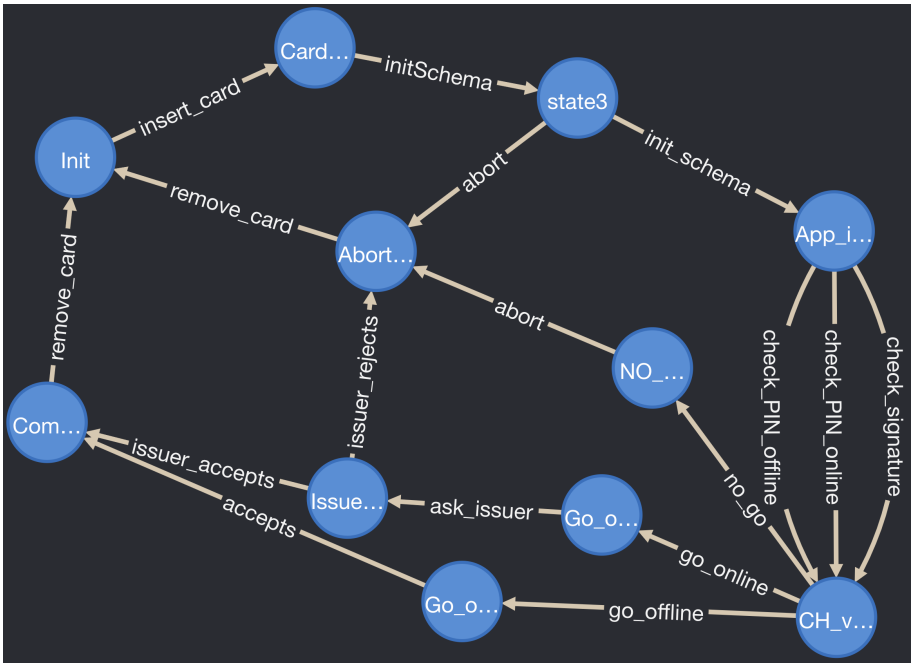

Figure 7.11: Card Payment Terminal FTS [61]

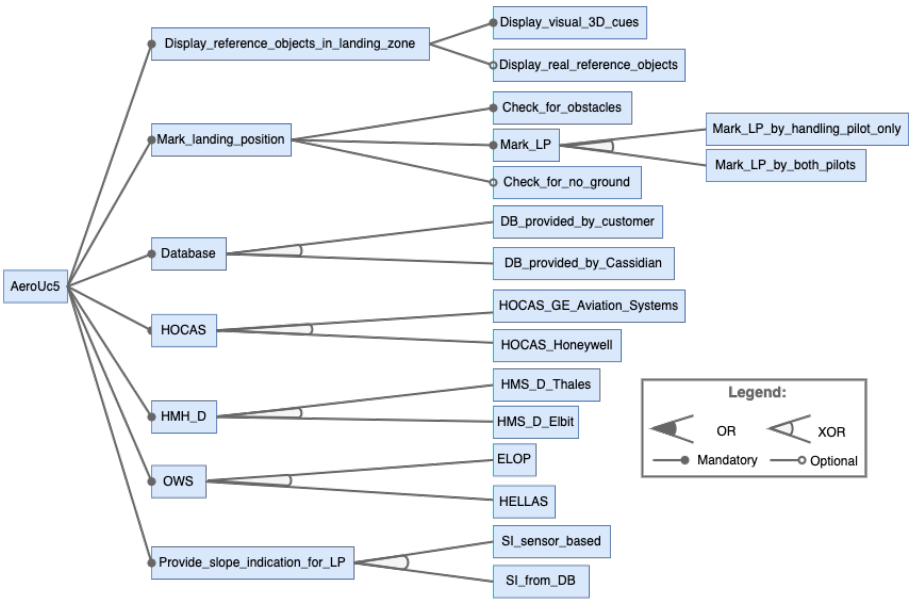

Figure 7.12: Sferion™ landing symbology function FM [61]

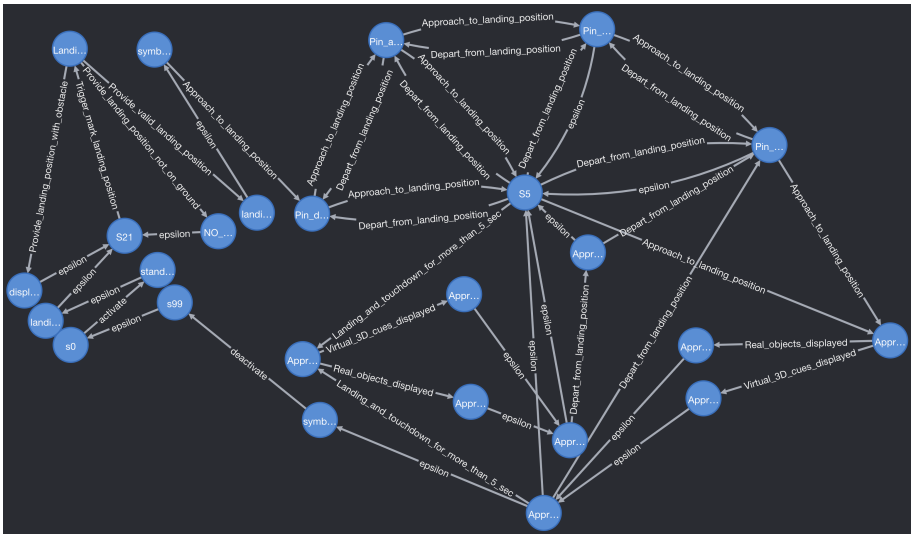

Figure 7.13: Sferion™ landing symbology function FTS [61]

CHAPTER 8

EMPIRICAL EVALUATION

This chapter is dedicated to the evaluation of the FL^* algorithm. We answer **RQ$_{1.1}$** to **RQ$_{1.4}$** and each of its sub-questions. We have executed LiFTS on the 5 datasets presented in Section 7.2 and computed different metrics. All experiments have been executed on a virtual machine with 16 Go of memory and 4 Intel CPU (2 core, 2 sockets).

Table 8.1 presents the characteristics of the final observation tables resulting from the execution of LiFTS. Columns contains the number of short featured prefixes and extended featured prefixes, before (Not S.) and after (Simp.) applying the simplification algorithm (Algorithm 5). The last column gives the number of suffixes (which is the same before and after simplification).

Table 8.2 presents the characteristics of the learned FDFA, for each case study before and after simplification (Not S. and Simp. respectively). Given characteristics are the number of states, the number of transitions, the average degree (*i.e.,* the average number of incoming/outgoing transitions per state) and the breadth-first search height (*i.e.,* the maximal number of states between the initial state and any other state).

Table 8.1: Characteristics of the learned OT for each case study

Model	**Prefixes**		**Ext. Prefixes**		**Suffixes**
	Not S.	Simp.	Not S.	Simp.	
Forum	8	8	70	0	7
SVM	73	24	1,029	0	18
Minepump	288	56	8,616	120	21
CP Terminal	98	40	1,891	32	20
Sferion™	129	129	1,420	228	47

Table 8.2: Characteristics of the learned FDFA for each case study

Model	States		Transitions		Avg. degree		BFS height	
	Not S.	Simp.	Not S.	Simp.	Not S.	Simp.	Not S.	Simp.
Forum	8	8	7	7	0.875	0.875	3	3
SVM	73	24	77	23	1.055	0.958	7	7
Minepump	282	56	369	58	1.309	1.036	9	9
CP Terminal	97	39	123	46	1.268	1.179	8	8
Sferion™	129	129	356	356	2.760	2.760	35	35

Table 8.3: Learning Times (in milliseconds)

Model	Setup	Learning	Simplification	Total
Forum	786	358	24	1,168
SVM	882	7,834	98	8,814
Minepump	436	528,127	311	528,874
CP Terminal	324	1,001,398	46,008	1,047,730
Sferion™	83,004	6,815,125	136,973	7,035,102

Execution times ($RQ_{1.1}$) We have measured the execution times, which are reported in Table 8.3. The *Setup* phase corresponds to the generation of counterexamples, as described in Section 7.1.1. In general, this phase takes less than a second, except for Sferion™, which takes about a minute. The *Learning* phase, during which the observation table and hypothesis automaton are constructed, varies in duration depending on the size of the system. The time required to find a valid hypothesis is less than a second for the forum case, 7.8 seconds for the soda vending machine, 8.8 minutes for the minepump, 16.69 minutes for the payment terminal and 113.59 minutes for Sferion™. For the smaller examples (forum, vending machine, and minepump), the simplification process takes less than a second, while for the card payment example, it takes 46 seconds, and for Sferion™, a little over 2 minutes.

Answer to $RQ_{1.1}$ (execution times): Execution times for the proposed framework vary depending on the specific case study being considered, ranging from just a few seconds to nearly 2 hours. These times are considered acceptable given the diversity of the case studies and the absence of any extraordinary execution infrastructure. Despite the variability in execution times, the framework is able to handle a range of examples within a reasonable timeframe, demonstrating its practicality and effectiveness in real-world scenarios.

Queries and Learning Rounds ($RQ_{1.2}$ and $RQ_{1.3}$) The number of membership queries for each case study, shown in Table 8.4, varies between 546 and 186,984. For equivalence queries, we have generated 200 positive and 200 negative examples, with a maximal depth (*a.k.a.* length threshold) of 35 actions. When we remove duplicates among produced traces, we obtain 7 to 363 examples. The number of learning rounds corresponds to the instances that we effectively need to address

Table 8.4: Characteristics of learning

Model	Counterexamples		Queries		Rounds	Resets
	Positive	Negative	MQ	EQ		
Forum	4	3	546	7	3	15,596
SVM	6	17	19,836	23	6	219,460
Minepump	19	35	186,984	54	11	2,934,811
CP Terminal	14	19	39,780	33	9	436,942
Sferion™	194	169	72,803	363	6	57,057,295

because they have not yet been modelled in the hypothesis.

The analysis of learning rounds in relation to the number of equivalence queries reveals the FL^* learning productivity. In the case of the Forum case study, the number of learning rounds is approximately half of the number of equivalence queries. This indicates that, on average, for every observed behaviour, LiFTS can infer a second distinct behaviour without the need for additional measures. Similarly, for the soda vending machine, the mine pump, and the card payment terminal, 70 to 80% of the total behaviour can be inferred this way. In Sferion™, this number increases up to 98% of inferred behaviour.

These findings underscore the effectiveness and efficiency of LiFTS in exploring the behaviour space of the system. By inferring additional behaviours from the observed ones, LiFTS demonstrates its capability to capture and model system dynamics without relying on extensive testing or exhaustive equivalence queries. This productivity is particularly valuable as it reduces the effort required to uncover a significant portion of the system's behaviours, providing valuable insights and contributing to the overall understanding of the system's behaviour.

Comparing the numbers obtained from our approach with those from state-of-the-art approaches [54, 56, 201] can be challenging due to an evaluation performed on different case studies, but mostly for the reasons mentioned in Section 6.5. To provide a more accurate evaluation of our algorithm, further analysis should be pursued. Conducting a comprehensive statistical analysis would enable a comparison between learning the products individually or as a family. This would involve comparing the learning curves for each case study, determining at what point family-based learning becomes more advantageous than product-based learning. By considering factors such as reusability and the associated costs of the family-based approach, we can determine the optimal approach for each scenario. Automating the computation of comparison metrics between the SUL and the learned model, such as Precision, Recall, and F1-score, would also be valuable. These metrics have been previously used by Damasceno et al. [56] to validate their approach and can provide a quantitative evaluation of the learned model's performance. By comparing these metrics, we can assess the accuracy and reliability of our approach in capturing the behaviour of the system.

By undertaking these additional analyses, we can obtain a more comprehensive and meaningful evaluation of our algorithm. This will facilitate a more accurate comparison with state-of-the-art approaches and provide deeper insights into the

strengths and limitations of our approach in relation to different evaluation criteria.

Answer to RQ$_{1.2}$ (membership queries): The number of membership queries ranges from 546 for the Forum case to 186,984 for Sferion™. Comparing the numbers with state-of-the-art approaches is challenging due to an evaluation on different case studies and other reasons discussed in Section 6.5.

Answer to RQ$_{1.3}$ (equivalence queries): For each case study, we executed 7 to 363 equivalence queries, depending on the VIS complexity. FL^* learning productivity is examined by comparing this number with the number of the learning rounds required to consider all the generated counterexamples. The results show that a significant portion (57% to 98%) of the system's behaviour can be inferred from previous observations, reducing the effort required for learning the system. To provide a comprehensive assessment of the algorithm's strengths and limitations, it is necessary to conduct a quantitative, statistical evaluation, allowing for a more thorough understanding of its performance across various metrics and in comparison to other approaches.

Resets (RQ$_{1.4}$) Resets are the nightmare of learning algorithms. They allow the system to get back to its original state, for example by restarting it. This is often mandatory to execute membership and equivalence queries from the right context. However, resets are expensive operations and may not be realistic for certain systems (*e.g.*, embedded system). As a result, minimising resets is typically a desirable goal.

In LiFTS, the number of resets for the evaluated case studies ranges from 15,000 to 57,100,000, as indicated in the last column of Table 8.4. Although these numbers may appear significant, it is important to note that the majority of these resets are a result of counterexample generation. Therefore, improving the counterexample generation process could have a significant impact on reducing the number of resets. This would make the approach more practical for systems where resets are expensive, contributing to the applicability of LiFTS in a wider range of domains.

Moreover, it is worth mentioning that for learning 15 sampled products of the minepump SPL (the only common system between the two approaches), Tavassoli *et al.* [201] reported 3,838,078 resets, while LiFTS achieved 2,934,811 resets. This comparison shows promising results and suggests that LiFTS is able to achieve a comparable or even lower number of resets for identical systems, which is encouraging.

Answer to RQ$_{1.4}$ (resets): The number of resets observed in our evaluation of the 6 case studies ranges from 15,000 to 57,100,000. By way of comparison, Tavassoli *et al.* [201] reported about 900,000 more resets than LiFTS when learning the minepump SPL in a product-based approach based on 15 sampled products. It is all the more encouraging that addressing counterexample generation has the potential to considerably decrease the number of resets.

CHAPTER 9

DISCUSSION

This chapter identifies threats to validity. Then, we suggest some new research direction. We conclude the chapter with a discussion about one specific hypothesis of FL^*.

9.1 Threats to Validity

Internal validity. To provide a more comprehensive evaluation of our approach, it is crucial to conduct a thorough statistical analysis to assess the impact of randomness, particularly in the order of counterexample generation. Additionally, evaluating the order of features given to the BDDs, which is currently determined by domain experts, is important. Furthermore, we should explore the effect of different parameters, such the number of positive and negative counterexamples (currently set to 200) and the length threshold for trace generation (set to 35).

We made the assumption that the FM was pre-existing, allowing us to focus on behavioural aspects. While this assumption may seem strong at first glance, we have shown that FM learning was a subject already treated in the literature. There exists several dedicated methods, such as natural language analysis [145], static analysis of variant configurators [191], variant catalogues (product tables), data mining [4, 5], evolutionary algorithms [152], *etc.*. Until now, we only consider systems with Boolean features. Learning systems with different types of features is still an open research question. Mapping the behaviour with the relevant structural information is another assumption we make (*i.e.,* associate specific products and behaviour). We develop that subject in Section 9.3.

For LiFTS, we chose Java as a programming language. This popular language allows us to interact with the VIBeS framework [62, 66] to reuse FTS utilities and with

the JavaBDD library [1] to manipulate feature expressions. Otherwise, we develop a completely new framework. Not relying on an existing framework, like *e.g.,* LearnLib [158, 179], implies more freedom in the architecture and a simpler code (fully dedicated to our task). However, we missed the generalisation of a popular framework, with many contributors. LearnLib proposes lots of optimisations, several implementations for different types of target models (*e.g.,* Mealy Machines, DFA, NFA, *etc.*), active and passive algorithms (*e.g.,* L^*, RPNI, *etc.*) and different types of equivalence queries (*e.g.,* conformance testing, search-based, *etc.*). It could be interesting, now that F^* is validated, to integrate it into such a framework.

Framework integration will also help optimising equivalence queries. First, we chose to focus on membership queries and the core of L^*, to ensure our algorithm was suitable. The way we implemented equivalence queries is thus very naive and could be improved. We could explore other techniques to generate counterexamples, like conformance testing or search-based. Adapting these techniques to take variability into account is one of our future research directions.

External Validity. Validating an automata learning approach can be quite challenging. The simpler way, used in most automata learning contest is to use a pre-existing model as a benchmark. The model is hidden and the result of each approach is compared to it. This is the method we used here to simulate equivalence queries. In practice however, the approach exhibits some disadvantages. First, it requires such a model which is not always easy to provide (especially in our case, see the dedicated challenge [202] to obtain similar benchmark). Moreover, the validation can be biased. If the learning approach is tuned specifically for one automata contest, it can pass all tests successfully but fail to learn a new model. In that case, the proposed learning approach lacks generalisation.

We consider models from various sources [44, 61], each possessing different characteristics such as the number of features, type of feature constraints, number of states, transitions from the original FTS, *etc.*. While we cannot guarantee generalisation to all product lines, a benchmark comprising six diverse datasets (including five from the literature and one homemade dataset) seems reasonable to demonstrate the viability of our approach. However, it is crucial to extend our analysis to larger and more complex datasets, such as industrial cases and those involving non-Boolean features. Nonetheless, working with extensive input alphabets and large datasets may not be feasible in the near future. To quote *Frits Vaandrager* [216]:

> "As a result, model learning currently can only be applied if there are less than, say, 100 inputs."
>
> — *Frits Vaandrager* [216]

This statement highlights the current limitations of model learning and reinforces the need for further research and advancements to handle larger and more complex datasets.

[1] https://github.com/com-github-javabdd/com.github.javabdd

To evaluate our approach in more realistic use cases, it is essential to extend the SUL interface by enabling interaction with a real system, rather than relying on a simulator. It would significantly enhance the realism and practicality of our approach, enabling us to assess its effectiveness in real-world scenarios. This extension would allow us to evaluate the accuracy and robustness of our method when confronted with real-time inputs, providing an opportunity to identify any limitations and refine our approach based on real-world observations. Overall, incorporating the SUL interface with a real system is a crucial step towards validating and enhancing the practical value of our approach.

9.2 Other Research Directions

Equivalence queries As mentioned previously, equivalence queries should be thoroughly studied in the context of variability. An equivalence query compares the hypothesis to the system or to a simulator of the system to check if they are equivalent. Usually, the simulator is defined as another hidden model, as we did in this thesis. The only possibility to decide if two models are truly equal is to use bi-simulation. Bi-simulation algorithms for FTS [23] are already very costly. Besides that, capitalise on a hidden model has two major flaws. Firstly, it is as good as the model corresponds to the real system. Secondly, it means having access to another behavioural model, which is exactly chat we want to learn in the first place. For all these reasons, hidden models are interesting in development process (as we did here), but are not realistic in a concrete case.

Therefore, we can approximate the equivalence by interacting with the system, through counterexamples. Counterexamples define the way we travel through the (very large, sometimes infinite) space of behaviour. Since it is not possible to explore the full state space, it is thus mandatory to have a good prioritisation technique, allowing us to explore a maximum state space in a minimum number of counterexamples. In practice, counterexamples can be efficiently obtained through conformance testing and monitoring, but these techniques have not yet been studied for variability contexts.

Framework optimisations To evaluate the effectiveness of FL^*, we conducted experiments using SPL with a limited number of features and actions. In order to apply the algorithm to larger benchmarks, improvements to LiFTS are necessary. While the current implementation performs acceptably within acceptable time limits (all case studies were learned in less than 2 hours), it is likely that it will not scale well for larger systems. One possible optimisation would be to leverage parallelism to expedite the learning process. However, considering integration into another framework offers additional advantages. By integrating a widely-used framework like LearnLib [179], we can benefit from better generalisation, code optimisations, and the development of additional functionalities, such as enhanced equivalence queries. Figure 9.1 presents the various learning options implemented within LearnLib. Furthermore, integrating with LearnLib would enhance the visibility of the algorithm,

attracting interest from researchers beyond the SPL community and potentially get advice from a wider audience.

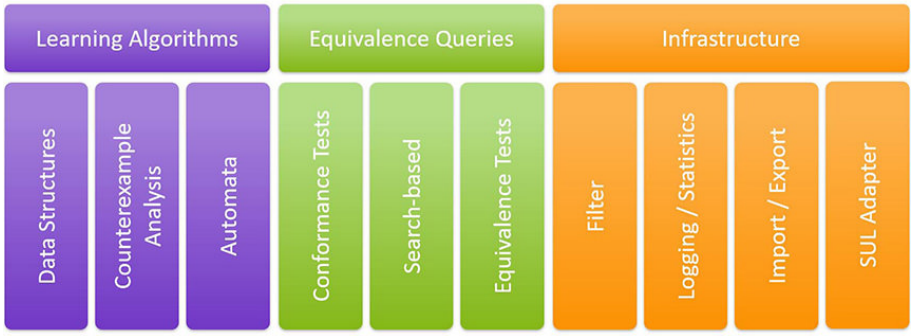

Figure 9.1: Principal features of LearnLib [179]

In addition to execution times, another crucial aspect to consider is memory usage. Currently, the observation table is stored in memory throughout the entire learning process. However, when dealing with large systems, the size of the table can become a limiting factor. To address this issue, we are interested in exploring alternative approaches, such as using external database storage. By leveraging an external database, we can potentially increase the model's resilience and enhance its learning capacities. Storing the observation table in a database allows for efficient management of large-scale data, enabling more scalable and robust learning processes.

Combining with FM Learning Instead of relying on a pre-existing FM, we could investigate learning structural and behavioural aspects jointly. Models are often outdated, if at all existing. Since FMs are not an exception to the rule, this scenario is probably more realistic. Moreover, by reusing the same artefacts, we could benefit from scale economies. Yet, the road is not straightforward and the potential costs could be superior to the benefits we derive from it.

9.3 Structural vs. Behavioural Variability

In the previous chapters, we provide an algorithm that learns an FDFA for a complete SPL. This algorithm is based on Angluin's L^* approach and treated feature expressions as first-class citizens. We modify the classical data structures to take variability into account and we rely on modified queries. However, this approach requires a strong hypothesis: **we assume that behaviour can be associated to structural variability** (*i.e.,* feature expressions). More precisely, the Teacher should be able to tell what are the products that can and cannot accept a given word (*i.e.,* if the product automaton accepts the corresponding run). This requirement is mandatory for both membership and equivalence queries. This hypothesis is not trivial and should be explored carefully.

First, we suggest a simple static mapping. In that scenario, each action from the input alphabet could be statically associated with a feature or a feature expression. This simple solution seems however too restrictive: what if we have the same action, with different constraints depending on **dynamic** constraints? Or a feature expression which is specific to a sequence of actions rather than a unique action? In consequence, we decided that a static mapping seemed too restrictive for such an inherently dynamic task as behaviour learning.

Then, we consider the specific ordering of actions to map feature to behaviour. Our goal was to keep the same hypothesis we used for developing FL^*. Thus, we can use the FM of the system and can generate and/or run execution traces. Quite naturally, we turned to supervised machine learning techniques. In particular, the domain of NLP brought our interest. In NLP, **the order of events** (*e.g.,* the order of words in a sentence) is of crucial importance. Like natural language, execution logs follow a kind of grammar and use a specific vocabulary (*i.e.,* the actions permitted by the system). Thus, in the last part of this thesis (Part III), **we apply NLP techniques to learn a mapping between features and execution traces**.

Part III

VaryMinions

CHAPTER 10

VARYMINIONS OVERVIEW

In the context of maintaining VISs, such as business processes and SPLs, examining behaviours is essential for accurate diagnosis. When managing multiple process variants, it becomes essential to identify the subset of variants that directly relate to the behaviour being investigated. This knowledge allows us to concentrate our analysis efforts on the relevant variants, thereby streamlining the diagnostic process.

In this part, we consider process/product executions stored in event logs, where an **event trace** (or trace) is an ordered sequence of events. Event logs do not usually contain information about which specific variant (or set of variants) could have produced each event trace. However, being able to locate variations is an essential part of any reengineering endeavour [15] (*e.g.,* debugging an anomalous execution or exploring refactoring opportunities). Identifying which variant(s) may have produced a given trace is also naturally useful for testing techniques, notably to sample which variants should be tested [107]. However, existing **variant analysis** [204] techniques do not answer this question but rather cover the inverse operation, *i.e.,* focusing on the differences between identified variants.

To support these activities, we train Recurrent Neural Networks (RNNs) [182, 187] architectures with different hyperparameters (loss and activation functions among others) to predict the candidate variant(s) that could have produced a given event trace. We make the following contributions:

(i) the first family-based approach, which we called VaryMinions, to map execution traces to variants of a system. We showed empirically that VaryMinions can distinguish 50 variants from 5,000+ event traces per variant;
(ii) a detailed account on the usage of LSTMs [119] and GRUs [41], two RNN architectures, on six different datasets, describing business processes and course

management system variants, showing that we can identify the variant(s) producing an event trace with high accuracy (> 80%);

(iii) four datasets openly available and based on Claroline [61, 64, 65] and containing $2 * 10$ and $2 * 50$ configurations with 5,000 traces per configurations;

(iv) a characterisation of the learning difficulty based on the behaviour shared amongst event traces.

(v) an implementation of our approach using two common Python frameworks reusing RNNs implementations (namely Tensorflow [59] and Keras [42]) and its evaluation on the different datasets.

In this chapter, we motivate the need for a tool like VaryMinions, then we give an overview of the proposed solution. Chapter 11 presents the datasets and the experimental setup with more details. Chapter 12 gives the results of our evaluation. Finally, Chapter 13 discusses certain factors influencing our experiments, such as hyperparameter variability and alternate labelling of variants.

10.1 Motivation

Two problems motivated the design of VaryMinions: i) locating bugs in VISs; ii) reverse engineering the behaviour of an entire SPL.

10.1.1 Testing and Bug Localisation Techniques

Modern SPLs incorporate many options to match the specific user needs. They have a common code base which interacts with the optional features' implementation depending on user needs. A high number of options prevents testing all the possible configurations of modern SPLs. For instance, the estimated number of configurations of the Linux kernel is about $2^{6,000}$. This number is orders of magnitude beyond the estimated number of particles in the universe (2^{320}).

Let us consider the following case study to illustrate the difficulty of testing VIS. *JHipster* is a configurable framework allowing to build full-stack web applications [184]. Halin *et al.* studied JHipster version 3.6.1, providing a feature model composed of 48 features and set up an exhaustive testing environment to build all the 26,256 configurations of this version [107]. This effort took eight man-months of engineering and more than 3,500 CPU hours of computations. Additionally, the authors interviewed JHipster's developers who acknowledged that they only fully tested 10 configurations. To select these 10 configurations, they focused on the most popular ones. Yet, 10 configurations could not cover the feature interaction bugs. This experiment shows that even when the system's size is modest enough for exhaustive coverage, the cost is prohibitive, especially for open-source communities.

A limited testing budget for huge configuration spaces implies that logs are necessarily incomplete and may miss important information for debugging, such as the mapping between traces and configurations. Besides the variability dimension, storing detailed information on the system's behaviour is costly. Cândido *et al.* [35] reported that logging preventively such detailed information would end up in large

logs (several terabytes) hindering analysis. However, to understand interaction bugs, we must observe the configurations impacted by these bugs. Predicting which configurations belong to a given trace gives the context for bug understanding while keeping a manageable log size.

10.1.2 Behaviour Reverse-engineering

In Part II, we have introduced a new adaptation of Angluin's algorithm, known as FL^*, to consider variability aspects. The FL^* algorithm focuses on learning a VIS in a family-based fashion. However, to accomplish this, it requires relating Angluin's queries and counterexamples to the configurations of the system. While the Teacher only knows about previously observed behaviours, it must be able to associate a new, unseen trace with the corresponding subset of system variants. As a result, a prediction technique is required to establish the relationship between unseen behaviours and the relevant system's variants (or their associated features, as discussed in Section 13.3).

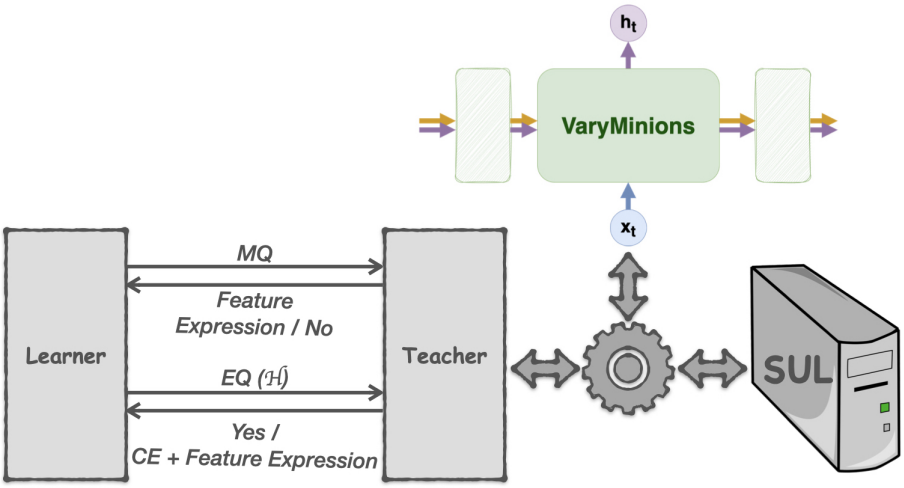

Figure 10.1: LiFTS overview: interaction with VaryMinions

In the core framework of this thesis, as illustrated in Figure 10.1, VaryMinions plays a key role by directly interacting with both the SUL, to make predictions, and the Teacher, to handle queries and counterexamples. Interactions can be separated into two scenarios:

(i) In the first scenario, instead of generating separate membership queries for each system variant, FL^* formulates a single query and determines the subset of variants that accept this query. In this scenario, VaryMinions takes the query as input and returns the corresponding subset to the Teacher.
(ii) The second scenario involves equivalence queries. The Teacher provides a counterexample for a specific variant. In this case, VaryMinions generalises

the counterexample to encompass all configurations that may be related to the counterexample, completing the Teacher's answer.

10.1.3 VaryMinions' Research Question

These examples illustrate the need to map execution traces (present in logs or derived on-the-fly via active learning) to VIS configurations. More precisely, given a set of event logs from different variants of a VIS, **how can we classify previously unseen traces to multiple variants?** To answer this question (**RQ$_2$**), we propose VaryMinions, an implementation described in the next section.

10.2 Architecture

With VaryMinions, our objective is to establish a mapping between execution traces, which represent a series of events occurring in a specific order, and system configurations capable of generating these traces. In this context, events are not randomly generated but rather depend on the previous succession of events. Some events are directly influenced by the preceding events, while others may be influenced by events that occurred much earlier in the trace. Chapter 3 demonstrated the effectiveness of LSTMs and GRUs as efficient text classifiers (*e.g.,* [135, 151]), highlighting their potential to handle textual data and ordered sequences of events. Notably, these architectures address issues like exploding or vanishing gradients that traditional RNN models suffer from [43, 118], making them well-suited for our task. Although their application in the domain of technical documents or software specifications is still relatively unexplored, LSTM and GRU architectures are well-suited for associating behaviours with variants. Furthermore, considering the sequential dependency of events, we treat traces as text, interpreting them as ordered sequences of symbols that adhere to a specific grammar.

Figure 10.2 provides an overview of VaryMinions' architecture. The input data *(a)* are a set of available execution traces. For training, traces are associated with the set of system variants that can produce them. The inputs first pass through an Embedding layer *(b)* that transforms the sequences of events into a vector of indexes (to make the representation more compact and to ease their processing). The embedding layer creates a structured space in which indexes that occur in a similar context are close. In this new representation space, indexes become vectors, and initial traces become tensors composed of numerical weights. This homogeneous representation allows performing mathematical operations on those weights through the rest of the network.

The embedding layer is also configurable, *e.g.*, we need to specify the number of dimensions of the representation space and the number of dimensions in the output tensors. We keep the number of dimensions the same in input and output to avoid combining different input dimensions into one output dimension. We then link this layer to the RNN layer *(c)*, which is instantiated with either LSTM or GRU units to learn the relationships between elements of the tensors. Again, this layer is

configurable, in particular with the number and usable kinds of units (detailed in Section 11.3).

There exist unidirectional and bidirectional [187] units. Unidirectional layers only consider the processing of the sequence in one direction (from start to end). In contrast, bidirectional layers also handle the other direction (from end to start), which can be helpful in language processing. In our case, traces are fully available at training time. Reading them forward and backwards can help grasp long-term relations between events. Because of our analogy with text, we use bidirectional units [187] only.

Then the network continues with one Dense layer *(d)* preparing for classification. We made the number of units in this layer the same as the number of classes (*i.e.*, configurations). The output of the network *(e)* is a vector of 1s and 0s whose number of elements is equal to the number of configurations of the system. This vector classifies the trace into one or more configuration(s). In this vector, 1s state that associated configurations can generate the input trace and 0s that they cannot.

For instance, let us take a simple system with three configurations. The output vector is thus of size three. If our prediction model outputs the vector $[1,1,1]$, it predicts that all the configurations can execute the input trace. In another case, the output vector is $[0,1,0]$. Then, only the second configuration is able to produce the input trace, *etc.*. One should note that our models cannot provide the output vector $[0,0,0]$ since the RNN selects at least the configuration with the highest score.

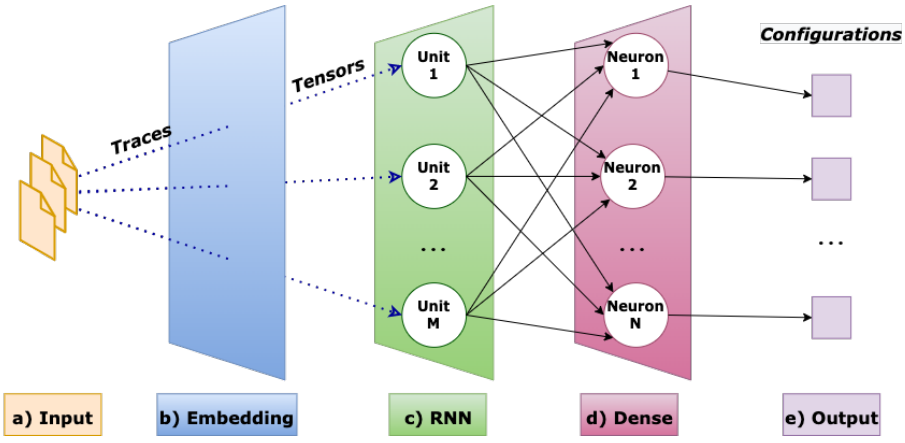

Figure 10.2: Description of the VaryMinions architecture

Unlike FL^* (Part II), VaryMinions does not necessarily require a complete feature model but rather a set of variants, which can be sampled from a feature model or obtained through product descriptions.

CHAPTER 11

EVALUATION PROTOCOL AND IMPLEMENTATION

In the following, we describe our evaluation protocol to validate that we can learn which variants of a system may have produced an execution trace. First, we state the research questions that drive this experimentation before describing the creation and annotation of our datasets. Then, we explain how we instantiated VaryMinions regarding our specific context. Finally, we present the running setup and the evaluation metrics.

11.1 Research Questions

We state the following research questions concerning the multi-classification of execution traces among the different VIS variants (**RQ_2**):

- **$RQ_{2.1}$ How accurately can we identify process variants based on their traces?** This question addresses the efficiency of our approach. To the best of our knowledge, this is the first attempt to use RNNs to learn such a mapping, we cannot compare it with the state of the art. Instead, we expect the RNNs to be at least better than random classifiers (accuracy higher than $> 50\%$).
- **$RQ_{2.2}$ What is the performance of LSTMs versus that of GRUs for process traces classification?** We would like to know which model architecture is the most appropriate for this task, if any.

11.2 Datasets Selection and Preprocessing

We use six different datasets that we divide into two groups. The first group contains the 2015 and 2020 editions of the Business Process Intelligence Challenge (BPIC).

Each dataset contains event logs, describing different executions of configurable processes:

- **BPIC15 (DS1)** represents building permit applications in five municipalities, each one corresponding to a process variant [222]; and
- **BPIC20 (DS2)** gathers data from the travel reimbursement process at the Eindhoven University of Technology (TU/e), where variants correspond to different kinds of documents to be managed [223].

The second group consists of four datasets containing event logs describing executions of different variants of **Claroline** [61, 64], an online course management system used at the University of Namur until 2018. Claroline was the main communication channel between students and lecturers, with approximately 7,000 users. Its architecture is plugin-based. Depending on needs, one can deploy new variants at runtime.

11.2.1 Business Process Intelligence Challenge (BPIC)

The original BPIC datasets (from [222, 223]) contain only valid and complete traces, together with other information. We prune the logs to keep only the process variant id, the trace id and the sequence of events. To cope with different trace lengths, we apply padding (*i.e.,* filling traces with other meaningless events and using a mask to know where the processing should stop). Trace duplicates are removed and since multiple variants can produce the same trace, we encode the variants into a binary vector (where the size matches the number of variants) that serves as a label. A value of one at the i-th index of the vector denotes that we observed at least once the trace associated with variant i. Traces associated with all variants have thus a vector full of ones. In the end, each trace is associated with one or more variants (*i.e.,* classes) and we expect the RNN models to learn these associations to predict the variant(s) for an unlabelled trace. We wrote this preprocessing procedure in Python and it is part of VaryMinions' implementation [87].

As described in Table 11.1, DS1 contains 5,542 traces after preprocessing, with a maximum of 154 events per trace. The five process variants are fairly equally represented since they contain 1,108 traces on average, with a minimum of 828 and a maximum of 1,350. DS1 is therefore well balanced. DS2 contains 2,074 traces after preprocessing, with 5 process variants and a maximum of 90 events per trace. The least and most represented process variants contain 89 and 1,478 traces respectively, with an average of 415 traces per variant. Therefore, the dataset is imbalanced, suggesting it is more difficult to learn accurately.

To better characterise the learning complexity, Table 11.1 shows the number of traces per class (*i.e.,* variant) and the overlap (*i.e.,* percentage of variant-specific and shared behaviour) between classes. The number of traces provides a first indication of the learning difficulty: more traces generally yield a more accurate network once trained. DS1 contains equally represented classes with limited overlap ($< 0.5\%$ in the last column), while DS2 is less balanced in how classes are represented and how they are interleaved, denoting a shared behaviour between multiple variants. In

Table 11.1: Overview of the preprocessed datasets used in our experiments. Class-specific metrics (cols 3–5) represent: (i) the number of traces per class, (ii) the percentage of traces assigned specifically to this variant in the dataset, and (iii) the percentage of traces shared by this variant and at least another one.

Dataset	Class Id	# Traces	% Variant-specific behaviour	% Shared behaviour
DS1 (BPIC15) 5,542 traces	**Munic. 1**	1170	99.658	0.342
	Munic. 2	828	99.638	0.362
	Munic. 3	1350	99.778	0.222
	Munic. 4	1049	99.905	0.095
	Munic. 5	1153	99.827	0.173
DS2 (BPIC20) 2,074 traces	**Int'l Decl.**	753	30.013	69.987
	Dom. Decl.	99	100	0.0
	Permit Req.	1478	64.344	35.656
	Prepaid	202	90.099	9.901
	Req. For Pay.	89	77.528	22.472
DS3 (Clar. Dis. 10) 50,000 traces	**Variant 1**	5000	100	0
	...	...	...	...
	Variant 10	5000	100	0
DS4 (Clar. Rand. 10) 50,000 traces	**Variant 1**	5000	100	0
	...	...	...	...
	Variant 10	5000	100	0
DS5 (Clar. Dis. 50) 250,000 traces	**Variant 1**	5000	100	0
	...	...	...	...
	Variant 50	5000	100	0
DS6 (Clar. Rand. 50) 250,000 traces	**Variant 1**	5000	100	0
	...	...	...	...
	Variant 50	5000	100	0

particular, for DS2, there is a big overlap between the *International Declaration* and the *Permit Request* variants, and between the *Prepaid Travel Cost* and the *Request For Payment* variants, while the *Domestic Declaration* variant is completely separated.

11.2.2 Claroline

Claroline is a highly configurable web-based system whose behaviour depends on a set of activated options. In total, Claroline contains 44 options leading to more than 5,406,700 unique variants. Handling such a large configurable system is not trivial as it requires deriving different variants and executing them in various ways to trigger different behaviours and collect, format, and process the corresponding event logs. Setting up such pipelines is hard and outside the scope of this thesis. For those reasons, we decided, instead of executing the actual system, to simulate executions of different variants using an FTS capturing the behaviours of different configurations of Claroline. The FTS was reverse-engineered by Devroey *et al.* [61,64]

from a 5.26 Go Apache web-server log containing 45,210,987 entries collected from January 2013 to September 2013 using a bigram inference method. The final FTS consists of 106 states and 2,053 transitions.

Simulations. The simulation of a given Claroline configuration works as follows. First, the FTS is **projected on the configuration** (*i.e.,* pruned) to keep only the subset of behaviours that can effectively be executed by the configuration. The result of that process is a classical transition system, describing a subset of the behaviours of Claroline. Second, the traces associated with the configuration are produced using random walks through the transition system. We generated 5,000 traces per configuration. To avoid infinite traces (*e.g.,* in case of a loop in the transition system), we also limited the size of a trace to 300 events. We relied on VIBeS [62, 66], a model-based testing tool for highly-configurable systems, to project the FTS and generate the traces.

We relied on two different strategies to select the different simulated Claroline configurations: random selection and dissimilarity-based selection. The random selection consists in selecting a set of (valid) configurations using a dedicated generator ensuring a random distribution of the selection. In our case, we used CMSGen [101], a fast uniform-like sampler. CMSGen comes with a default parameterisation that we reused as is.[1] Unlike random, dissimilarity-based selection [113] picks configurations in such a way that they are as dissimilar as possible when considering their selected options. For our evaluation, we used PLEDGE [114], a search-based dissimilarity-driven configuration selection tool. We selected the default parameterisation of PLEDGE, with one minute per generation. We have set the number of configurations to two different values: 10 and 50. This way, we can go beyond the difficulty provided by the BPIC datasets and check that our method can run when the number of configurations is higher. While 50 is still small compared with the number of possible unique variants of Claroline (*i.e.,* $> 5,000,000$), it is closer to a realistic setting.

Event logs datasets. We have derived the four different event logs datasets based on the following sets of configurations of Claroline:

- **Claroline Dissimilar 10 (DS3)** regroups execution traces of 10 different configurations of Claroline, selecting the most dissimilar sets of options. This dataset should lead to more discriminated traces and better classifications.
- **Claroline Random 10 (DS4)** gathers traces from 10 different instances of Claroline, randomly chosen to have a more realistic dataset.
- **Claroline Dissimilar 50 (DS5)** is similar to DS3, but with 50 configurations to allow more diversity.
- **Claroline Random 50 (DS6)** is similar to DS4, but with 50 configurations.

For each of these datasets (DS3 to DS6), the output of this generation process is a file containing 5,000 traces per configuration that we can use as an input for

[1] See https://github.com/meelgroup/cmsgen for details.

VaryMinions. In our case, we thus have either 50,000 traces per file (for 10 configurations) or 250,000 traces per file (for 50 configurations), as shown in Table 11.1. The last two columns of this table show systematically 100% of variant-specific behaviour and 0% of shared behaviour for Claroline datasets, meaning that for each trace, at least one action is specific to one variant of Claroline. This is due to the use of a sampler for selecting the configurations, giving very little control over the traces overlap. Due to the huge amount of possible variants (*i.e.,* $> 5,000,000$), the chance to find any shared behaviour between multiple variants is almost zero.

11.3 RNN Parameterisations

As we said before, because we use sequences of events, we investigate the use of RNNs to learn to which configuration(s) we can associate a trace. More specifically, we focus on LSTMs and GRUs. As for many DL models, hyperparameters must be defined. Because there are so many, we decided to vary only a few of them to try to understand how much impact they may have on learning. We focused on the functions that are used inside the networks and that may impact the quality of the predictions. We also manually selected a subset of hyperparameters that we fixed to a specific standard value. Hyperparameters and their values are described in detail hereafter and summed up in Table 11.2.

Number of hidden layers. One specific aspect that impacts the learning capabilities of neural networks is their topology. Since the traces are short compared to text documents, we decided to use networks with only one hidden layer. It may avoid potential overfit, that can emerge from more complex structures (*e.g.,* auto-encoder), while offering satisfactory prediction performance.

Units. In earlier experiments [86], we used different numbers of units regarding the RNN layer *(c)*. This number affects the topology of the network and may help to grasp more complex concepts if this number increases. Yet, having too many units on a layer may lead to dealing with redundant information that will deteriorate the final prediction performance of the network [93]. On the contrary, a layer with a smaller number of units may not have the capability to grasp interesting information which may also harm the prediction performance [93]. Based on our previous experiences, we decide to set the number of units to 30 which has shown relatively good performances while limiting the training time.

Training set, batch size and epochs. Other hyperparameters can be set affecting the training time and the optimisation of the many different parameters (*e.g.,* weights between layers and units) of the networks. Common hyperparameters to set are the ratio of data used to train the model and those used to evaluate the performance of the model; the size of the batch of data that the model will have to deal with during training, which may mitigate overfitting; and the number of time the model will optimise parameters over the whole training set (*i.e.,* the number of epochs). Each of these hyperparameters was set as follows:

(i) the percentage of data used for training is set to 66% of the whole dataset which is a common value in the community, the remaining traces are used in the test set to assess the generalisation performances of the trained models;
(ii) we set the batch size to 128, which is adapted to the dataset size;
(iii) we set the number of epochs to 20 to avoid overfitting. In our preliminary evaluations (evaluated between 10 and 50 epochs), a plateau was reached after approximately 15 epochs. We finally set the number of epochs to 20, to allow for small increases in accuracy.

Activation functions. Activation functions are defined at the level of units (*i.e.,* neurons) and respond to an input signal. If the signal is strong enough, the neuron is activated and the output is also high. Though different activation functions can be used for each neuron, it is common to define an activation function for an entire layer. We have used a Rectified Linear Unit (ReLU) function on the hidden layer RNN layer (*i.e., (c)* in Figure 10.2) to alleviate the vanishing gradient problem. Regarding the Dense layer *(d)*, we experimented with two common activation functions which are sigmoid and hyperbolic tangent (tanh). Both are shown in Figure 11.1. The main difference between both is their definition domain which affects how they handle negative input values. The sigmoid function is defined over $[0;1]$ meaning that as the values get closer to $-\infty$ the neuron is closer to being non-activated at all (*i.e.,* the output signal is 0) while as the input values are getting larger the response is also getting larger. When the input value is 0, the response is 0.5. On the other hand, tanh is defined over $[-1;1]$. It may be useful to take into account negative correlations and when the input value is 0, the response is also 0. Using one or the other may affect the "strength" of the signal that will reach the last layer for classification in turn affecting which class (*i.e.,* configuration) will be recognised.

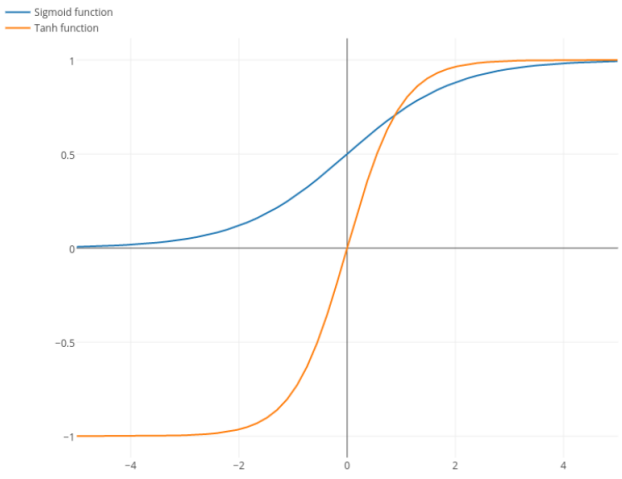

Figure 11.1: Sigmoid (blue) and tanh (orange) function responses represented by the Y-axis depending on the input signal (X-axis).

Loss functions. Loss functions are used during training to optimise the weights of the networks by back-propagating errors. We have used three loss functions already implemented in tensorflow [2], namely Binary Cross-Entropy (with and without logits, respectively named hereafter Bin-CE and Bin-CE logits) and the Mean Squared Error (MSE). Logit is defined as the inverse function of the sigmoid. We also implemented two custom loss functions: a variant of the Jaccard distance [124] (named Weight_-Jaccard hereafter), and the Manhattan distance between two vectors. The motivation for these two last functions is that because a single trace might be assigned to different process variants, thus the error should be defined considering a comparison of elements of vectors but not from a single value. This difference between two vectors should define a distance score. The Manhattan distance (sometimes called L1 norm) computes the sum of absolute differences between each element of the two vectors (*i.e.,* , in this case, the process variants). The Jaccard distance assesses how many equal elements of two vectors are over their size. We have implemented a variant of the Jaccard distance to cope with floating-point values generated by the networks. The Jaccard distance was employed to evaluate trace dissimilarity in variability-intensive systems (*e.g.,* [67]). Further discussions about the use and characteristics of these loss functions are provided in Section 13.2.

Table 11.2: Hyperparameters settings

Hyperparameter	Considered values
Type of Classifier	GRU, LSTM
# Units	30
% Training Set	66%
Batch Size	128
# Epochs	20
Activation Function	sigmoid, tanh
Loss Function	Bin-CE, Bin-CE logits, MSE, Weight_Jaccard, Manhattan

11.4 Model Training

We decided to use only a training set and a test set in our evaluation due to the number of available execution traces. The training and performance evaluation process is done as follows: i) the entire dataset is randomly split into training and test sets. We have used the Keras function *train_test_split* [3] that ensures the data distribution of classes among the two sets are similar. ii) a model is trained using the training set. iii) its prediction performances are evaluated on the test set. To mitigate biases in our analyses we decided to train and evaluate the performance of each parameterisation ten times on each dataset. For each run, the whole training and performance evaluation process is started again (*i.e.,* splitting into training and

[2] https://www.tensorflow.org/api_docs/python/tf/keras/losses

[3] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

test sets, training the model, and evaluating its performance). The fact that the splits are done each time mitigates the chances to train and evaluate a model on the best sets solely. Not only that it may change the data used for training the model but it may change the order of appearance too, which may have an impact on the trained model.

11.5 Evaluation Metrics

This work is the first attempt to use RNNs to classify execution traces among variants of a system. One of our goals is to evaluate if such a DL technique is appropriate for this task. We thus computed four different standard metrics that are **Accuracy**, **Precision**, **Recall**, and **F1-score**.

Accuracy. To evaluate the quality of the models that have been learnt, the usual metric is the **Accuracy** measure. Accuracy is defined as

$$Acc = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions} \tag{11.1}$$

It is a standard measure in the ML community to assess how well a model performs from a high-level point of view. It has the advantage to be easily computable and it can also be used to refer to the number of wrong predictions (*i.e.,* $1 - Accuracy$).

However, when classes are not well balanced (*i.e.,* the number of traces is way more important for at least one class than for others), **Accuracy** may hide some important information such as the number of correct predictions for the classes with more data may take the lead on the number of wrong predictions of the others resulting in a high ratio. To mitigate this aspect from our analysis, we only consider other measures.

Precision. One usual metric to account for the performance of a prediction model is its **precision**. It can be calculated for each class as follows:

$$Precision = \frac{Number\ of\ correct\ predictions}{Number\ of\ predictions\ for\ the\ class} \tag{11.2}$$

where $Number\ of\ predictions\ for\ the\ class$ is the number of correct predictions and the number of additional data that are wrongly predicted to belong to the class (*i.e.,* false positives).

We gathered all these individual **precision** measures into a global one using a weighted average:

$$Prec = \frac{\sum_{i=1}^{c} Precision_i * supp_i}{Number\ of\ data} \tag{11.3}$$

where c is the number of classes, $Precision_i$ the **precision** measure for class i, and $supp_i$ the number of data with label i.

Recall. Similarly to the **precision**, the **recall** is also standard to report on the predictions of a model. It can also be calculated for each class and it is defined as follows:

$$Recall = \frac{Number\ of\ correct\ predictions}{Number\ of\ labelled\ data\ for\ the\ class} \tag{11.4}$$

where $Number\ of\ labelled\ data\ for\ the\ class$ is the number of data labelled with the class under consideration.

Similarly to the **precision**, we computed a weighted average to get an overall **recall** measure for the model:

$$Rec = \frac{\sum_{i=1}^{c} Recall_i * supp_i}{Number\ of\ data} \tag{11.5}$$

where c is the number of classes, $Recall_i$ the **recall** measure for class i, and $supp_i$ the number of data with label i.

F1-score. The F1-score is obtained through the harmonic mean of **precision** and **recall** to get an overview of the global performance of the model in one single measure. The **F1-score** in the case of two classes is defined as:

$$F1\text{-}score = 2\frac{precision * recall}{precision + recall} \tag{11.6}$$

Again, we can apply this calculation on each class and average with a weight equal to the proportion of data of each class in the (test) set to get an overall value for the model. The three last metrics were computed by the **precision_recall_fscore_support** [4] function in Scikit Learn before being averaged. Also, we computed confusion matrices [5] for each class. They are available in our replication package.

11.6 Running Infrastructure

Finally, Table 11.2 shows: 2 $models \times 1\ \#units \times 1\ \%training\ set \times 1\ batch\ size \times 1\ \#\ epochs \times 2\ activation\ funtions \times 5\ loss\ functions = 20$ different parameterisations of RNNs. We conducted these experiments on three different HPC facilities hosted by the CÉCI.[6] On the first cluster, called Dragon1, we used 1 CPU with 8 cores per task (Intel Sandy Bridge, E5-2650 processors at 2.00GHz) with a Tesla Kepler accelerator (K20m, 1.1 Tflops, float64). For runs on Dragon2, we used 1 CPU with 12 cores (Intel SkyLake, Xeon 6126 processors at 2.60 GHz) associated with an NVidia Tesla Volta V100 accelerator (5120 CUDA Cores, 16GB HBM2, 7.5 TFlops, double precision). On the third cluster, Hercules, we had access to 1 CPU with 8 cores per task (Intel Sandy Bridge, Xeon E5-2660 processors at 2.20 GHz) with an NVidia GeForce accelerator (RTX 2080 Ti, 7.5 TFlops, double precision). Each CPU

[4] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_fscore_support.html

[5] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_fscore_support.html

[6] http://www.ceci-hpc.be/

has been allocated 3 GB of RAM. All our scripts are written in Python 3, with the Keras and Tensorflow frameworks for deep learning. In total, we ran our 20 different network parameterisations with 10 repetitions on the six different datasets, resulting in $20 \times 10 \times 6 = 1,200$ runs and more than 151 days of execution. The time needed for a single execution varies between 44 seconds and 13 hours depending on the dataset and the GPU type.

CHAPTER 12

EVALUATION RESULTS

In this chapter, we answer our two research questions separately based on:

- box-plots presented in Figures 12.1 to 12.4, showing accuracy, precision, recall and F1-score for each parameterisation of each dataset;
- a multi-comparison statistical analysis (see Figure 12.5), using Friedman's test with Nemenyi's post-hoc analysis;
- Tables presented in Appendix C, with average and standard deviation for the four computed metrics.

All the results (*i.e.,* for each execution of each parameterisation) are also available in our replication package [87], including the code to compute the metrics, box-plots and statistical tests.

12.1 Performance ($RQ_{2.1}$)

Table 12.1: Number of RNN parameterisations reaching predefined accuracy thresholds. We take into account 120 parameterisations. Accuracies are averaged over 10 runs on each dataset. Each cell indicates the number of times a given RNN model type (column) reaches the threshold (row). The last column gives the total (LSTM+GRU) per accuracy range.

Accuracy	LSTM	GRU	Total
accuracy > 70%	23	21	44
50 < accuracy < 70%	8	7	15
accuracy < 50%	29	32	61

Table 12.1 reports the averaged accuracy (over 10 runs) of the 20 considered parameterisations of RNNs, over the 6 datasets (*i.e.,* 120 models). The columns group results based on the used model (LSTM or GRU), while the rows group them depending on the average accuracies. We define three categories according to a predefined threshold: i) below 50% where we consider models as performing worst than a random assignment to system variants and thus useless; ii) between 50% and 70% where we consider models as being slightly better than random assignments; iii) over 70% where we consider the models as performing well. Out of the 120 models, 44 RNNs parameterisations (first row) yield an accuracy higher than 70%, 15 are between 50% and 70%, and the remaining 61 have an accuracy below 50%. It means that nearly half of the considered models perform better than a random guess, a majority of which (*i.e.,* 44 parameterisations out of 59%) performs well in our context.

The highest averaged accuracy for datasets BPIC15 and BPIC20 (top of Figure 12.1, or Tables C.1 and C.2 in Appendix) is 88% and 87% respectively with high stability (*i.e.,* low standard deviation). On BPIC20, only five parameterisations out of twenty do not reach 50%. Even better, for BPIC15 only five parameterisations are lower than 70% of accuracy. Top of Figures 12.2, 12.3 and 12.4 confirm these results by giving similar values for precision, recall and F1-score respectively.

Despite the size complexity of the Claroline datasets, at least one parameterisation obtains an averaged accuracy of 80% for each dataset. For Claroline Dissimilar 10 (middle left of Figure 12.1 and Table C.3 in Appendix), the top parameterisation reaches 99.6% and 4 different parameterisations are above 85%. Claroline Random 10 and Random 50 (middle and bottom right of Figure 12.1, or Tables C.4 and C.6 in Appendix) also have several parameterisations above 80%, and their top one gets over 95% of accuracy. Claroline Dissimilar 50 (bottom left of Figure 12.1, or Table C.5 in Appendix) has only one row with an averaged accuracy of 80% and only two other rows above 70%. Among the remaining, 15 rows are below 30%.

Note that, for Claroline Dissimilar 50, boxplots are either spread out or centred on low values (bottom left of Figure 12.1). Moreover, the top three rows also report a high standard deviation for the accuracy (*i.e.,* higher than 0.13 and up to 0.38, in Table C.5 in Appendix). It highlights that the results lack stability: at least one execution out of ten does not belong to the same values range. Regarding Claroline Random 10 and Claroline Random 50 (middle and bottom right of Figure 12.1), the top three parameterisations show very compact boxplots with few outliers. This suggests a more stable accuracy, as confirmed by a standard deviation between 0.03 and 0.11 for the accuracy (Table C.4 and Table C.6 in Appendix). The top two parameterisations of Claroline Dissimilar 10 (Table C.3) both show an accuracy higher than 0.99 and a standard deviation lower than 0.001, demonstrating very stable results.

Overall, the number of configurations of the Claroline system (10 or 50) neither influences averaged accuracy nor the standard deviation. Similarly, how we sample (random-based or dissimilarity-based) configurations does not impact accuracy. As for BPIC15 and BPIC20, the other metrics (precision, recall and F1-score presented respectively in Figure 12.2, 12.3 and 12.4) only confirm this analysis as they follow

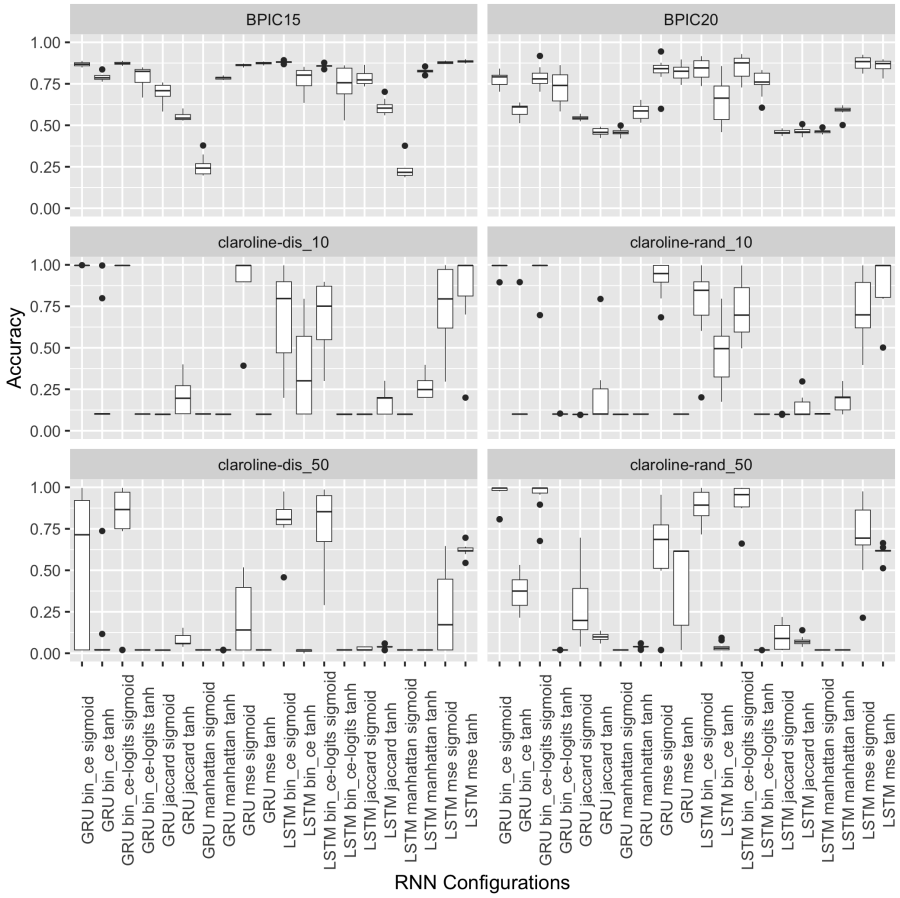

Figure 12.1: Boxplots showing the Accuracy over 10 runs for each parametrisation of each dataset.

the same tendencies.

Answer to $RQ_{2.1}$ (performance): we were able to train RNNs providing an accuracy above 70% (and even above 80%) for each dataset. On Claroline Dissimilar-10 the accuracy can reach 99.6%. The associated standard deviations can be small (*i.e.,* < 0.01) but they are usually higher with the Claroline datasets, regardless of the number of configurations used or the way we select them. Yet, these results suggest there is potential to use RNNs to automatically classify newly generated execution traces among the variants of a system rather than trying to do it manually.

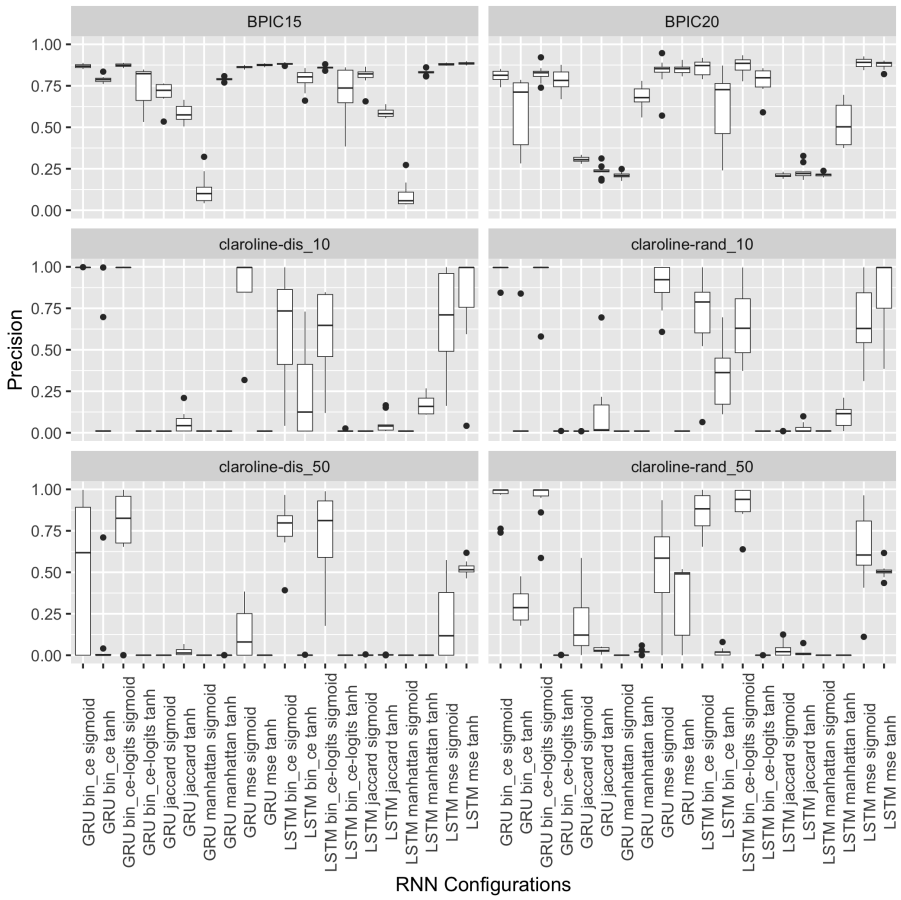

Figure 12.2: Boxplots showing the Precision over 10 runs for each parametrisation of each dataset.

12.2 LSTM vs. GRU ($RQ_{2.2}$)

Our second RQ is about the prevalence of each type of RNN. Can LSTM or GRU be considered better and should it be preferred in this context? To answer this question, we hypothesise that one kind of RNN prevails over the other one and perform a multi-comparison statistical analysis of each 20 RNN parameterisations on all 6 datasets. We used Friedman's non-parametric test [92] with a significance level $\alpha = 0.05$. This test ranks parameterisations over accuracy and then determines if the differences between parameterisations are significant. We further complete this result with Nemenyi's post-hoc procedure [125, 166] indicating the statistical differences between parameterisations. This procedure can determine equivalence classes, regrouping parameterisations that are statistically similar in regarding accuracy.

Figure 12.5 show the results of Nemenyi's test. After executing Friedman's test, we obtain a p-value under 0.001, meaning that there is a statistical difference between

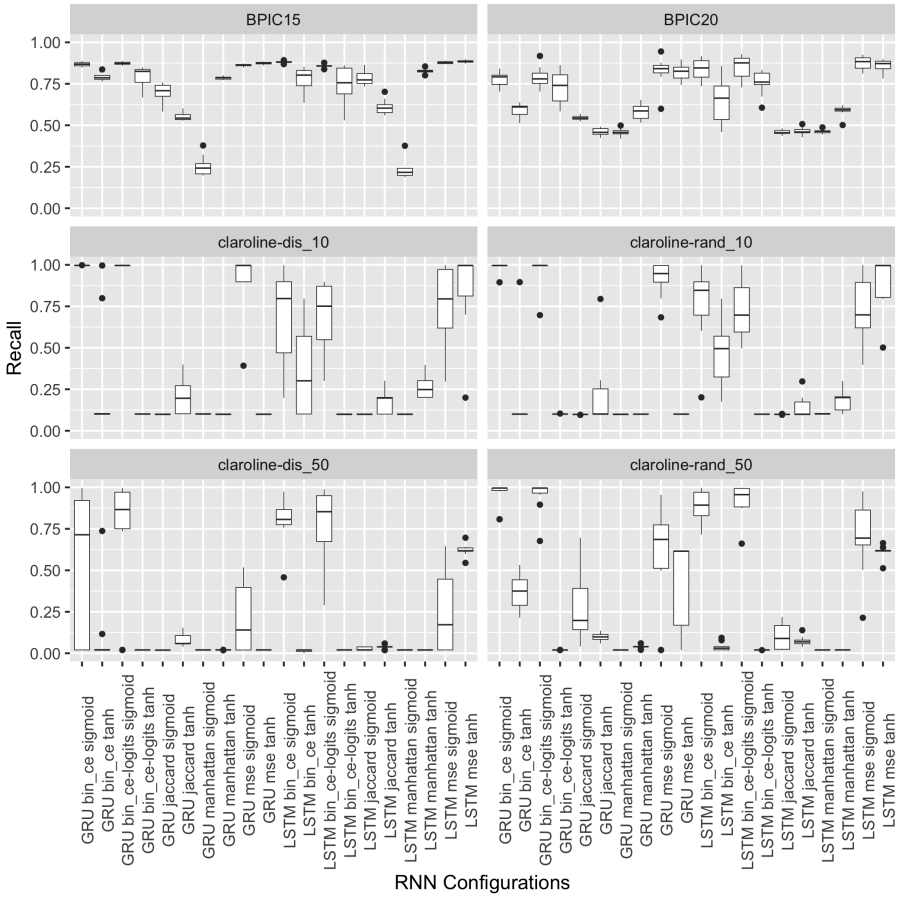

Figure 12.3: Boxplots showing the Recall over 10 runs for each parametrisation of each dataset.

the accuracy of some of the parameterisations. Nemenyi's post-hoc procedure shows that the minimum distance between two statistically different groups of parameterisations (*i.e.,* the critical distance) is 3.828. The bottom of Figure 12.5 shows the seven best parameterisations over all the datasets. Statistically, they are equivalent and perform better than the remaining ones (belonging to a different group).

Four pairs of loss and activation functions out of ten seem to stand out from the test. They are:

- MSE and sigmoid
- binary cross-entropy and sigmoid
- binary cross-entropy with logits values and sigmoid
- MSE and tanh (with LSTM only)

For most datasets, these parameterisations can predict the right set of variants

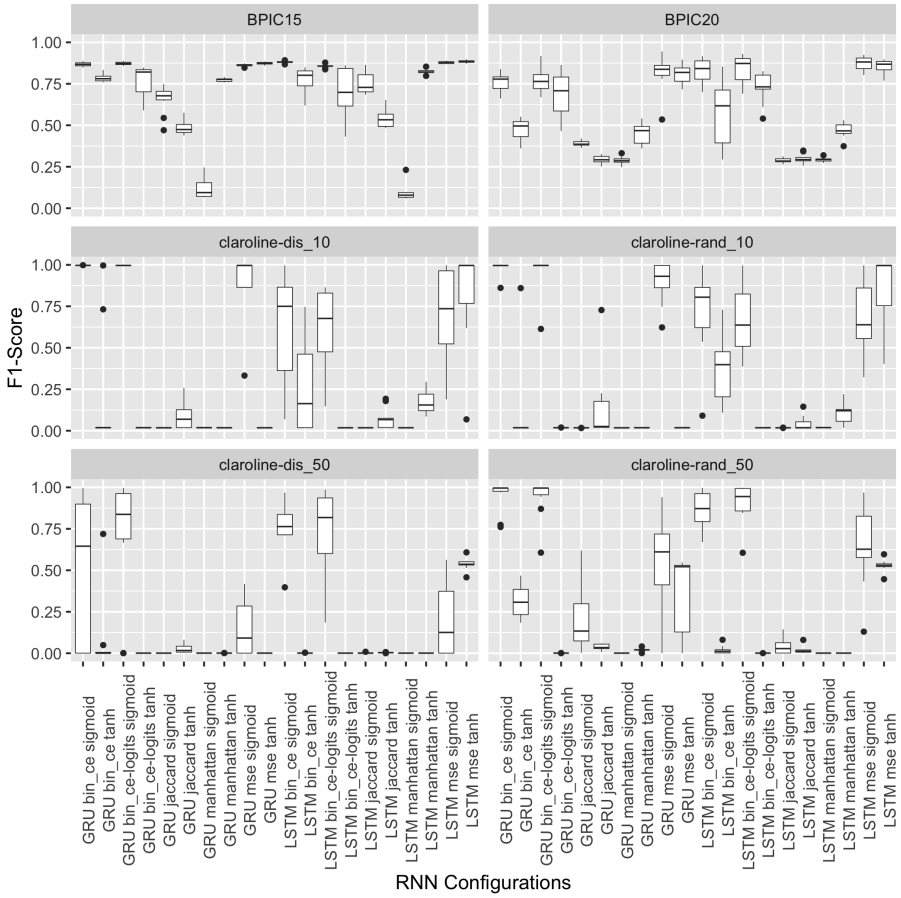

Figure 12.4: Boxplots showing the F1-Score over 10 runs for each parametrisation of each dataset.

with an accuracy greater than (or very close to) 70% (confirmed by Figure 12.1 and Appendix C). However, sometimes a combination also gives bad results. This is the case for MSE and sigmoid, both with LSTM and GRU, where accuracy does not exceed 0.25 for Claroline Dissimilar 50 (Table C.5).

We can observe that the dedicated loss functions (Manhattan and Jaccard distance) give discouraging results compared to the other "classical" loss functions. Nemenyi's procedure (Figure 12.5) assigns them the highest mean ranks. For all Claroline datasets, the accuracy is always under 30%. On BPIC15 and BPIC20, they give better results (respectively up to 82% for BPIC15 and up to 58% for BPIC20) but still lower than the other loss functions.

Regarding the activation functions, our statistical analysis shows that 6 out of the 7 best parameterisations use sigmoid instead of tanh.

Nemenyi's procedure shows that LSTMs are present in 4 of the top parameterisations and GRUs in 3 of them. However, these parameterisations are indistinguishable

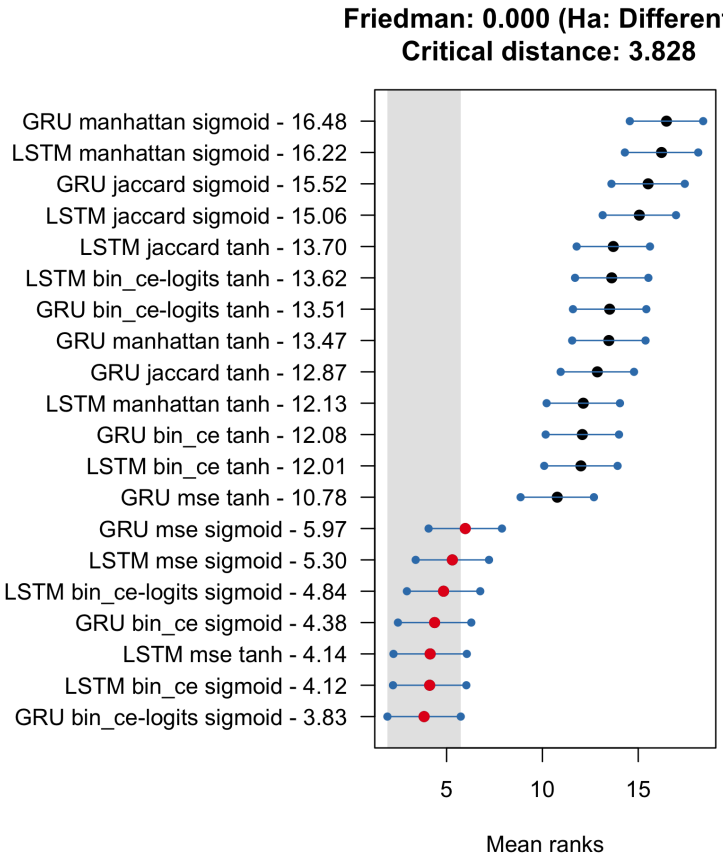

Figure 12.5: Result of Friedman's statistical test along with Nemenyi's post-hoc analysis over all datasets and parameterisations

regarding accuracy (*i.e.*, critical distance < 3.823). Appendix C shows that the best parameterisation is an LSTM for BPIC15, BPIC20 and Claroline Dissimilar 50, but it is a GRU for the three other datasets. GRU is also the model giving the best accuracy amongst all datasets with up to 99,6% for Claroline Dissimilar 10 (Table C.3). Moreover, the count of LSTMs and GRUs in each category of Table 12.1 shows similar numbers and indicates that using GRU or LSTM does not influence the results.

Answer to $RQ_{2.2}$ (classifiers): In the top combinations of all six datasets, we observed a mixed performance of LSTMs and GRUs, with no absolute winner. A statistical comparison showed that 4 out of 7 parameterisations use LSTMs, without any significant difference between the 3 parameterisations using GRUs. Moreover, GRU gives better results on 3 of the datasets (Claroline Dissimilar 10, Claroline Random 10 and Claroline Random 50). Hence, we cannot conclude the prevalence of one over the other for these six datasets. Moreover, our results suggest using the sigmoid activation functions rather than tanh.

CHAPTER 13

DISCUSSION

This chapter discusses threats to validity that we identified and other aspects driving our future work.

13.1 Threats to Validity

Internal validity. The datasets we used contain clean and consistent traces (*i.e.,* they omit inconsistent traces when the system crashes or an unexpected event occurs). The BPIC community ensures this property for BPIC15 [222] and BPIC20 [223]. For the Claroline datasets, trace consistency is ensured by construct, withe the use of an FTS model and the VIBeS framework [62, 66] as a trace generator (for Claroline). For a new VIS, a preprocessing step should take care of trace consistency (*i.e.,* a trace should capture a complete user session). It does not entail that the dataset captures the whole system's behaviour. Indeed logs and models inferred from them represent a partial view of it.

The deep learning community is very active, producing new types (or combinations of types) of models every few months, especially for image processing tasks, where competition is fierce. It is less so regarding models dedicated to time sequences. We selected LTSMs and GRUs for their ability to deal with temporal sequences and to evade the vanishing or exploding gradient issue.

We evaluated 20 distinct parameterisations of RNNs over six datasets. We designed them regarding our goal, based on our previous work [86]. However, since exhaustive coverage of the hyperparameter space is impossible, we may have missed some relevant parameterisations. Dealing with the inherent variability of hyperparameters is a research challenge *per se*.

A way to optimise the parameterisations is to use hyperparameter tuning techniques such as random search or auto-ML [164]. We did not use any in this work

but tried to scope the parameterisation space with a manual approach similar to a grid search approach [86]. One motivation for this choice is that VaryMinions is the first effort to use RNNs to classify execution traces for variants of systems. Thus, we were not primarily interested in finding the best-performing model (aka the goal of hyperparameter tuning). Rather, we show that, with reasonable effort, finding a well-performing RNN model parameterisation is possible.

External Validity. Although our method applies to two different application domains (SPLs and configurable processes), we cannot ensure that it generalises to all types of VIS. We used six different datasets having different characteristics to mitigate the risk that our method may work only on simple datasets. Among the ones we have used, some were taken from existing competitions (BPIC), and some were generated from scratch (Claroline) allowing us to vary and control the complexity of the learning by modifying the amount of traces available and/or the number of configurations to deal with. Let us note that reverse-engineered models from logs necessarily form an incomplete representation of the behaviour of the system. Indeed, logs cannot capture all execution traces that are often infinite for any real-world system. Besides, we do not guarantee that our cases cover the whole spectrum of VIS, given their diversity and spread.

A problem when using DL techniques in such a context is imbalanced representations in the training set. The training set may contain fewer occurrences of a configuration of a system or a process (*e.g.,* because of lower popularity or fewer actions need to be performed) with the risk that the trained model may neglect classification errors involving these configurations since they can be considered as rare events. While the Claroline datasets were generated in such a way that imbalance representations were limited, we had no control over the BPIC datasets. They exhibit configuration imbalance but our RNN models coped with it (*i.e.,* successfully classified traces belonging to these configurations). Thus, we took no further actions to mitigate this risk. Of course, class imbalance impact is case-specific.

13.2 Hyperparameter Variability

The use of RNNs in this context requires carefully dimensioning the network and considering many parameterisations that can influence classification performances. In what follows, we discuss two elements that may influence them.

Loss functions. We use the mean squared error (MSE) to evaluate prediction errors while training a network, which is traditionally preferred when tackling a regression problem. However, Hui and Belkin [122] showed that this assumption lacks solid theoretical foundations and that MSE is suitable for classification. In particular for NLP applications, where MSE usually outperforms cross-entropy.

The choice of the loss function is tricky since we need to take care of multiple aspects: the formalisation of the problem (*e.g.,* single or multi-label, regression or classification) or the way to compute errors. Even when trying to choose the loss function according to these points (*e.g.,* Jaccard distances have been used to

solve SPL problems, as in [67]), our results indicate that the MSE works surprisingly well. Given the importance of a loss function on the observed performance, experimenting with additional loss functions appears promising. For example, the focal loss [149], which penalises more misclassified instances than well-classified ones, offers a perspective that we aim to follow.

The interplay of Losses and Activations. We deliberately chose to explore custom loss rather than activation functions. Loss functions are easier to adapt to the problem at hand (by quantifying how far we are from the true label) acting on the network output. Yet, activation functions and loss functions have distinct roles in the network, and they should be considered complementary and not independent. Both are important in the learning process. Activation functions come after every layer inside the network and, together with the weights, set the importance of a specific neuron through the propagation of the network. Loss functions are defined at the end of the network and are used to provide the final class(es). Loss functions are also used to back-propagate the classification errors through the network to optimise the weights in the training phase. From this short description, it is clear that activation and loss functions' interactions affect the model performance. The former may block or lower the importance of discriminative information if incorrectly set while the latter defines the distance from the labels, from which the network optimises itself. Hence, assessing the impact of one type of function alone is not possible. Further investigations on which combinations would be best suited are needed. Defining new custom activation functions for this specific context is a possible option.

Complexity of the neural networks We argued that learning a trace-to-variant mapping was feasible due to the number of traces *w.r.t.* the limited number of process variants. Generally, the challenge lies in the fact that having temporal sequences forces dependencies between elements that are usually learned separately. We suppose that deeper RNNs (*i.e.,* increasing the number of hidden layers) may have a positive impact. Adding more layers increases the complexity of the model (as well as requiring more resources for training), but allows for a more accurate mapping between traces and variants. Yet, the risk of overfitting must not be neglected. In the future, we will also consider architectures such as auto-encoders to produce a compact intern representation of traces, that could be more efficient in discriminating them according to the process variants. Similarly to other application domains (*e.g.,* image or speech processing), learning more compact representations could rely on new feature descriptors instead of only considering events of a trace.

13.3 Variant-based vs. Option-based Labelling

Our results indicate that applying classification techniques on a variant-based approach (*i.e.,* identify the variants producing a specific trace) using RNNs is promising. However, it has a major drawback: being able to predict that a trace is generated by a variant requires seeing at least one (usually much more) trace(s) generated from this variant. Said differently, enumerating all the variants and executing them all

at least once is required for further predictions. If in our evaluation the number of variants was limited, the combinatorial explosion problem inherent to VISs may prevent us from applying these techniques to larger configurable processes like, for instance, continuous integration workflows with hundreds of options, leading to an intractable number of possible variants.

One future possibility to address this limitation is to work on data representation. Indeed, a variant is formed by a combination of (Boolean) options, corresponding to a **configuration** of the system. If we cannot enumerate variants, enumerating options is possible. In this case, we need a new representation which can depict the three states of each option: activated, deactivated or undetermined (*i.e.,* the presence of the option is not relevant for the current context). The use of three-valued logic to represent valid products in SPLs and its consequences have already been studied, for instance in [21]. The neural network will learn a partial configuration allowing for a more fine-grained mapping. This would be useful to locate precisely a combination of options yielding a given anomalous event trace. One can use such learned models in fault localisation and repair techniques [77].

13.4 Data Availability

As for any DL technique, the issue of data availability is also present in this work. We managed to train our models with few execution traces (*i.e.,* thousands) compared to the potentially infinite number of traces that the considered systems can produce. However, VaryMinions remains a supervised machine-learning technique and requires a set of execution logs, labelled with the variants of the system that have produced them.

To reduce the labelling effort, the recent field of **semi-supervised learning** [40] techniques seems interesting. Semi-supervised learning takes place when, in the training set, some data have labels but a majority of them are unlabelled (*e.g.,* due to the prohibitive cost of labelling). The goal is thus to learn a model while being able to label **automatically** the unlabelled data. In this area, label propagation [123, 142] automatically assigns a new label via propagating the label of already known similar data. We envision using the same technique (or an adapted version) to reduce the labelling effort while being able to take into account more and more execution logs which may improve the prediction performances of VaryMinions models.

Part IV

Postface

CHAPTER 14

CONCLUSION AND FUTURE RESEARCH DIRECTIONS

Quality assurance is a complex, yet essential, part of any software engineering endeavour. For VIS systems, the challenge becomes even more significant, due to the combinatorial explosion of the number of variants. To cope with variability efficiently, we can rely on family-based modelling approaches. While structural variability models, such as FM, can help achieve tasks related to product validity, verification and testing require a behavioural model of the SPL. Nonetheless, creating these models manually is a laborious and costly process, requiring a high level of expertise.

In this thesis, we present a model-based behavioural SPL learning framework. Our approach relies on an adaptation of L^* active automata learning algorithm, where feature expressions are treated as first-class citizens. Taking variability into account throughout the whole learning process allows scale economies compared to classical product-based approaches. However, this solution requires to build a mapping between features and behaviour. To address this challenge, we explore the use of RNNs to learn this mapping from a collection of annotated traces. By leveraging RNNs, we aim to automate and enhance the process of learning the intricate relationships between behaviour and their corresponding software variants within the VIS context.

14.1 Summary of Contributions

This thesis aims to study the relationship between VIS features and behaviour. We propose the LiFTS framework (Figure 14.1) to answer the two following research questions:

RQ$_1$ **How to automatically learn FTSs in order to ease the burden of modelling FTS and support continuous VIS QA activities?**

RQ$_2$ **How to classify previously unseen behaviour to multiple variants of a VIS?**

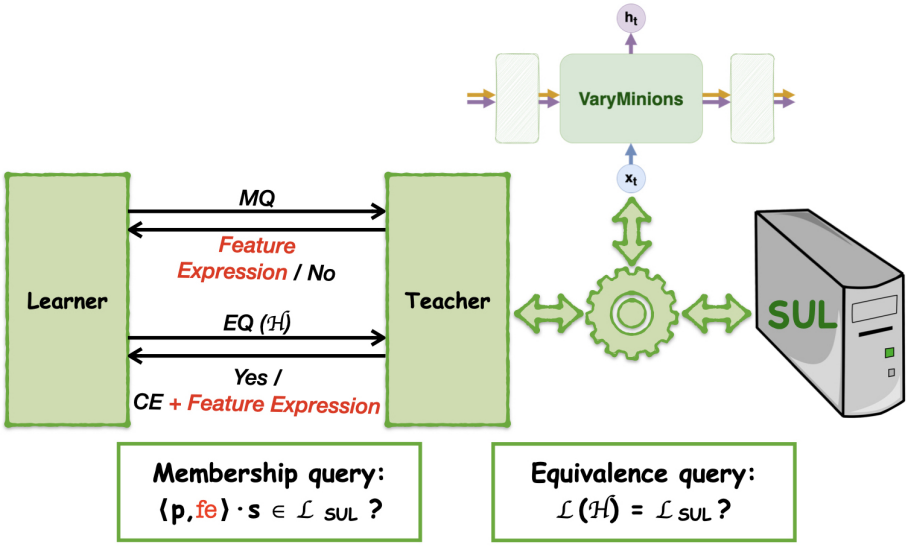

Figure 14.1: LiFTS General Framework

To better apprehend the challenges of these tasks, we address them separately and we propose the following improvements:

- **Variability-L*:** To address **RQ$_1$**, we introduce a novel model called Featured Deterministic Finite Automaton (FDFA), which serves as a finite and deterministic version of FTSs. We then present a new algorithm, $Featured-L^*$, that fundamentally differs from previous approaches. We also provide the implementation of FL^* called LiFTS, which successfully learns five distinct case studies within a short time frame ranging from a few seconds to less than two hours. Furthermore, we highlight the unique aspects of FL^* compared to previous approaches. Additionally, we define a visualisation aid for FTS/FDFA models.
- **VaryMinions:** To address **RQ$_2$**, we draw inspiration from NLP techniques. We establish an analogy between an NLP sentence and an execution trace, where words or actions are arranged into valid sentences based on a specific grammar. Leveraging this analogy, VaryMinions utilises two types of RNNs, namely Gated Recurrent Units (GRUs) and Long Short Term Memory (LSTM) networks. These RNN architectures enable the classification of VIS behaviour among different variants that can reproduce them. By utilising GRUs and LSTMs, which overcome the vanishing or exploding gradient problem, our multi-classification technique proves effective. We evaluate VaryMinions on six diverse datasets, encompassing both SPL and business process domains.

Each dataset consists of up to 50 variants and 5,000 event traces per variant. Throughout the evaluation process, we execute 120 distinct RNN parameterisations over approximately 150 days.

By combining VaryMinions to Variability-$\mathbf{L}^*$, we could ensure that the Teacher is able to predict which part of the VIS is concerned for each query and counterexample.

Replicability. To prevent potential replicability issues, we provide a replication package for each contribution, openly available online [83, 87].

14.2 Perspectives and Future Work

This section aims to present our perspectives and outline potential research directions for enhancing variability-aware automata learning and the association of behaviour with VIS configurations. By exploring these new challenges, we can ultimately advance the field and enable a more efficient and reliable development of VIS systems.

14.2.1 Integrated Approach

We aim to explore an integrated approach for the LiFTS project, which focuses on studying the relationship between VIS features and behaviour. While addressing the research questions individually has led to significant progress, there is an opportunity to further improve the effectiveness and efficiency of the overall approach by integrating key components together. Specifically, we propose investigating the joint learning of structural and behavioural aspects, as well as integrating VaryMinions into the LiFTS framework.

Integration of VaryMinions into LiFTS VaryMinions has demonstrated promising results in classifying VIS behaviour among different variants. Currently, the LiFTS framework relies solely on a FTS simulator of the SUL. However, our ultimate goal is to integrate VaryMinions directly into LiFTS, thereby enhancing the capabilities of the Teacher. By seamlessly integrating VaryMinions, we can leverage its advanced classification techniques and eradicate the unrealistic assumption of the existence of a hidden model of the SUL. By combining this integration with the establishment of a direct access to the SUL, the overall approach can achieve a more realistic and accurate analysis of VIS behaviour. To achieve this, we need to align the data representations, designing appropriate interfaces, and ensuring compatibility with the existing framework.

Learning Structural and Behavioural Aspects Jointly The current assumption of a pre-existing FM in LiFTS has limitations, as FMs are often outdated or non-existent. To address this, we propose investigating the joint learning of structural and behavioural aspects, enabling the approach to align with real-world scenarios.

By learning both structural and behavioural aspects from the same artefacts, we can achieve economies of scale and improve the accuracy of capturing system characteristics. However, learning structural and behavioural aspects jointly poses challenges such as developing suitable algorithms, handling complex feature dependencies, and ensuring scalability.

Development of a Comprehensive Platform To further enhance the capabilities of the LiFTS project, we envision the design of a complete platform that enables the generation, visualisation, and editing of FTSs. FeatureIDE [130] is a framework for feature-oriented software development, relying on FMs. We could consider a similar platform but focusing on the behavioural aspects rather than structural modelling. By providing comprehensive support for FTS manipulation, the platform would facilitate the modelling, analysis, and transformation of FTSs. This would empower researchers and practitioners to effectively work with FTSs, generate code from FTSs, and visualise the relationships between features and behaviour.

By integrating VaryMinions into the LiFTS framework and exploring the joint learning of structural and behavioural aspects, we anticipate significant advancements in capturing and analysing VIS behaviour. However, this integrated approach poses challenges that require careful consideration and further research. The potential benefits, including improved accuracy, scalability, and real-world applicability, justify the exploration of this integrated approach in future work. Additionally, the development of a comprehensive platform for FTS manipulation would provide a valuable resource for the research community and SPL engineers.

14.2.2 Scalability and Interfacing

The study of equivalence queries in the context of variability is essential for achieving a faster and more scalable implementation of FL^*, but it presents several challenges. When using a simulator, the most straightforward approach to determine the equivalence between a hypothesis and the SUL is bi-simulation. Yet, bi-simulation algorithms are computationally expensive. Furthermore, relying on a hidden model as a reference introduces additional flaws in the learning process. Requiring a model beforehand undermines the fundamental purpose of learning, since it implies that we already know the model we are targeting. Using a simulator is thus an acceptable alternative to check the correctness of our approach, but is not applicable in real-life scenarios. In practice, the learning process should be capable of accommodating without relying on pre-existing knowledge. To address these limitations, it becomes necessary to approximate equivalence through interactions with the system using counterexamples. However, efficiently exploring the vast space of behaviour requires effective prioritisation techniques. Currently, there is a lack of research on conformance testing and monitoring techniques specifically tailored to the context of variability, offering opportunities for future exploration and advancements in the field.

To assess the effectiveness and practicality of our approach in real-world scenarios, it is crucial to extend the SUL interface to interact directly with a real system, rather than relying solely on a simulator. This extension would provide a more realistic environment for evaluating our approach and allow us to gauge its accuracy and robustness when exposed to real-time inputs. By integrating the SUL interface with a real system, we can gain valuable insights into the limitations of our approach and make necessary refinements based on observations and feedback from real-world use cases. Ultimately, this step is vital for validating and enhancing the practical value and applicability of our approach in real-world settings.

In addition to execution times, memory usage is a significant consideration. Storing the observation table in memory can become problematic for large systems. Using external database storage may turn out to be a more scalable and robust solution. Storing the table in a database, allows us to handle the memory requirements more effectively and opens up possibilities for tackling larger and more complex systems.

Optimisations are necessary for applying the FL^* algorithm to larger benchmarks. Leveraging parallelism and integrating with widely-used frameworks like LearnLib [179] or Aalpy [163] would enhance performance, generalisation, and provide additional functionalities like enhanced equivalence queries.

Overall, these aspects offer promising directions for scaling LiFTS to larger variability-intensive systems.

14.2.3 Other Learning Approaches

In this thesis, our focus is on adapting Angluin's L^* algorithm [9] as the central learning approach to obtain a behavioural model of a VIS. However, it is worth noting that there exist other relevant learning approaches that could be considered for this task. Exploring these alternative approaches could potentially provide valuable insights and complement the effectiveness of our learning approach.

One such approach is grey-box learning, which combines elements from both white and black-box learning to leverage the advantages of each. Popular static analysis frameworks like Java Pathfinder [1], SootUp [2], or Infer [3] can be particularly valuable in this context, fostering the extraction of essential information from the source code. Shoham *et al.* [193] use static analysis to mine Internet API specifications. Fraser *et al.* [91] also use static analysis to infer object usage and thereby generate more meaningful tests. By delving into white-box learning, we can enhance the learned FTS structure with feature expressions and state variable values that are not easily accessible in a complete black-box scenario. For instance, by employing a mix of concrete and symbolic analysis, also known as concolic execution, we could extract valuable insights on variable assignments and interactions between features. Leveraging concolic execution would allow us to infer feature expressions, providing valuable insights into the intricate relationships between features and their impact

[1] https://github.com/javapathfinder/jpf-core/
[2] https://soot-oss.github.io/SootUp/
[3] https://fbinfer.com/

on system behaviour. Howar *et al.* use a mix of static, dynamic and concolic analysis to learn safe interfaces for critical embedded systems [120]. They also suggest a grey-box scenario where predicates or guards are exploited to guide black-box learning [121]. Moreover, white-box learning also facilitates the collection of additional information, such as probabilities, weights, and other relevant quantities, which can extend the FTS formalism to be more expressive and provide a more nuanced representation of system behaviour.

Another intriguing avenue is process mining, which involves tailoring the process mining workbench ProM [221, 224] to discover VIS models. ProM excels at discovering Petri net models from event logs without any a-priori information. As an open-source tool framework, ProM supports various process mining techniques through plug-ins and warmly welcomes contributions of new plug-ins. Buijs *et al.* [34] use genetic algorithms to combine process models mined from event logs of different variants, in a variant-based fashion. Adapting ProM to discover VIS models in a family-based perspective in mind could be an interesting avenue.

By exploring and integrating these other learning approaches, such as white/grey-box learning and process mining, we can enrich the analysis of VISs and gain a more comprehensive understanding of their behaviour. Therefore, future work should investigate the potential benefits of these alternative approaches and compare their effectiveness in learning variability-aware behavioural models. Such comparisons will contribute to a more thorough evaluation of the learning process and its applicability to different types of VISs.

14.2.4 Feature-based Mapping

Our research has shown promising results in using classification techniques with RNNs for trace-to-variant mapping in a variant-based approach (*i.e.,* when behaviour is related directly to the system variants). However, the requirement of observing traces from each variant poses a significant limitation, as it may be impractical to enumerate and execute all variants, especially in large-scale configurable processes.

To address this limitation, one potential future direction is to focus on data representation. Instead of enumerating variants, we can focus on enumerating features to describe the different system configurations. By representing each option with three states (activated, deactivated, or undetermined), we can train neural networks to learn partial configurations, allowing for a more fine-grained mapping. This approach enables precise identification of feature combinations that lead to specific behaviour. Such learned models can be valuable in fault localisation and repair techniques, where pinpointing the specific combination of features causing an issue is important.

By exploring alternative data representations and leveraging the flexibility of neural networks, we can potentially overcome the challenge of enumerating all variants and extend the applicability of classification techniques to larger VISs. This opens up opportunities to tackle the combinatorial explosion problem inherent in

such systems and enhance the analysis and prediction of system behaviour based on traces.

14.2.5 Neural Network Architecture

To optimise the classification performance in trace-to-variant mapping, it is crucial to carefully consider several aspects of RNNs. Specifically, attention should be given to the following factors:

- **Loss functions:** The choice of an appropriate loss function is critical for training the RNN model effectively. While MSE is commonly used for regression problems, it has also shown promising results for classification tasks, even outperforming cross-entropy in certain NLP applications. However, the exploration of alternative loss functions, such as the focal loss, which prioritises misclassified instances, holds potential for further enhancing performance. Experimenting with different loss functions can help determine the most suitable one for trace-to-variant mapping.
- **Activation functions:** Both activation functions and loss functions play distinct but complementary roles in the neural network. Activation functions determine the importance of neurons through network propagation, while loss functions quantify the distance from the true labels and optimise the network weights during training. The interaction between these functions influences the overall model performance, and their combination should be carefully considered. Assessing the impact of one type of function alone is not sufficient, and exploring the use of custom activation functions specific to this context is worth exploring.
- **Network complexity:** The complexity of the RNN architecture can significantly affect the model's ability to learn the trace-to-variant mapping accurately. Deeper RNNs with increased numbers of hidden layers can potentially capture more intricate dependencies and improve the mapping performance. However, increasing network complexity also comes with the risk of overfitting the data, where the model becomes too specialised to the training set and performs poorly on unseen data. Therefore, striking a balance between network complexity and generalisation capability is crucial. Additionally, exploring alternative architectures, such as auto-encoders, can provide compact representations of traces and enhance discrimination based on process variants.
- **Data Labelling:** To alleviate the labelling effort, we propose exploring the field of semi-supervised learning techniques. Semi-supervised learning is applicable when a training set contains labelled data, but the majority of the data remains unlabelled due to the high cost of labelling. The objective is to develop models that can automatically label the unlabelled data while learning from the labelled examples. Label propagation techniques can be employed to automatically assign labels to unlabelled data by propagating labels from similar known instances. By leveraging these techniques, we can reduce the labelling effort while incorporating a larger number of execution logs, potentially enhancing the prediction performance of VaryMinions models.

By carefully considering these aspects of RNNs researchers and practitioners can optimise the classification performance in trace-to-variant mapping tasks. It is essential to experiment with different configurations and conduct thorough evaluations to identify the most effective choices for improving the accuracy and robustness of the model.

14.3 Final Remarks

In 2010, Classen *et al.* [48] recognised the need for a formal verification technique tailored to software product lines. However, they faced a significant challenge due to the combinatorial explosion of variants in SPLs, where the number of potential configurations grows exponentially with the number of features. This made it difficult to model SPL behaviour and apply classical model checking techniques effectively. To address this issue, Classen *et al.* [48] introduced a novel formalism called FTS. FTSs represent SPL behaviour by employing transition systems labelled with feature expressions. This new approach allowed for more efficient modelling and analysis of SPL behaviour, overcoming the limitations posed by combinatorial explosion [48].

Several years later, from 2014, Devroey *et al.* [65] built upon the concept of FTSs and extended their application to the field of software testing. They proposed coverage criteria and mutation testing analysis techniques that leveraged FTSs to define test suites specifically designed for SPLs. This work demonstrated the practical benefits of using FTSs in the context of software testing and highlighted the potential for more advanced testing methodologies in SPLs.

Since their introduction, FTSs have made significant contributions to the field of SPLs. Their ability to address the combinatorial explosion challenge and provide a more efficient approach to modelling SPL behaviour has been widely acknowledged and appreciated by researchers and practitioners in the SPL community. FTSs have proven to be a valuable tool for formal verification [46, 48, 50, 51] and testing [63–66, 68] in the context of SPLs, enabling more effective analysis and quality assurance.

The recognition of the original paper with the Most Influential Paper award at SPLC 2020 reflects the impact and significance of FTSs in the SPL domain. It highlights the pioneering nature of the work and its lasting influence on subsequent research and developments in the field. The validation of FTSs by the SPL community underscores their importance as a fundamental technique for addressing the unique challenges posed by SPLs.

In the present work, we complete the cycle by introducing the missing piece in the process: a reengineering process to obtain the FTS model required for formal verification and testing techniques. Our contribution focuses on developing a systematic approach to derive the FTS model from existing artifacts, such as execution traces. This reengineering process closes the loop and enables the seamless integration of formal verification and testing techniques within the SPL development lifecycle.

By bridging the gap between formal verification, testing, and the reengineering process, LiFTS provides a comprehensive solution for addressing the challenges of modelling SPL behaviour, verifying its correctness, and ensuring effective testing.

This holistic approach enhances the reliability and quality of SPLs and opens up new avenues for research and advancements in the field of SPL engineering.

LEARNING A FORUM SPL

This appendix gives the different artefacts produced while learning the Forum SPL (described in Section 7.2.1).

A.1 Input Feature Model

```
c 1 Forum
c 2 AuthenticationMethod
c 3 Cancellation
c 4 Anonymous
c 5 RegisteredUser
p cnf 5 4
1 0
2 0
4 5 0
-3 1 0
```

Listing A.1: FM of the Forum case study in Dimacs format

A.2 Artefacts Obtained the First Learning Round

Figure A.1 is the observation table obtained after the first round of learning. The FDFA obtained is described by the Listing A.2 XML file and can be visualised as Figure A.2.

```
<?xml version="1.0" encoding="UTF-8"?>
```

table_0

OT	[ε]
(Q_0) [ε] - ((AuthenticationMethod && Anonymous && Forum) || (AuthenticationMethod && RegisteredUser && !Anonymous && Forum))	false
[abort] - ((AuthenticationMethod && Anonymous && Forum) || (AuthenticationMethod && RegisteredUser && !Anonymous && Forum))	false
[send_msg] - ((AuthenticationMethod && Anonymous && Forum) || (AuthenticationMethod && RegisteredUser && !Anonymous && Forum))	false
[log_in] - ((AuthenticationMethod && Anonymous && Forum) || (AuthenticationMethod && RegisteredUser && !Anonymous && Forum))	false
[choose_pseudo] - ((AuthenticationMethod && Anonymous && Forum) || (AuthenticationMethod && RegisteredUser && !Anonymous && Forum))	false
[enter_credential] - ((AuthenticationMethod && Anonymous && Forum) || (AuthenticationMethod && RegisteredUser && !Anonymous && Forum))	false

Figure A.1: Forum case study: observation table after the first learning round

```
2     <fts>
3         <start>Q_0</start>
4         <states>
5             <state id="Q_0"></state>
6         </states>
7     </fts>
```

Listing A.2: Forum case study: FDFA in XML format after the first learning round

Figure A.2: Forum case study: FDFA visualisation after the first learning round

A.3 Artefacts Obtained the Second Learning Round

After 1 counterexample, the table has evolved in Figure A.3. The FDFA is updated to Listing A.3 XML file, with its visualisation in Figure A.4.

table_1

OT	[ε]	[abort]	[send_msg]
(Q_0) [ε] - ((AuthenticationMethod && Anonymous && Forum) || (AuthenticationMethod && RegisteredUser && !Anonymous && Forum))	false	false	false
(Q_1) [choose_pseudo] - (AuthenticationMethod && Anonymous && Forum)	false	(AuthenticationMethod && Cancellation && Anonymous && Forum)	(AuthenticationMethod && Anonymous && Forum)
(Q_2) [choose_pseudo, send_msg] - (AuthenticationMethod && Anonymous && Forum)	(AuthenticationMethod && Anonymous && Forum)	false	false
(Q_3) [choose_pseudo, abort] - (AuthenticationMethod && Anonymous && Forum)	(AuthenticationMethod && Cancellation && Anonymous && Forum)	false	false
[abort] - ((AuthenticationMethod && Anonymous && Forum) || (AuthenticationMethod && RegisteredUser && !Anonymous && Forum))	false	false	false
[send_msg] - ((AuthenticationMethod && Anonymous && Forum) || (AuthenticationMethod && RegisteredUser && !Anonymous && Forum))	false	false	false
[log_in] - ((AuthenticationMethod && Anonymous && Forum) || (AuthenticationMethod && RegisteredUser && !Anonymous && Forum))	false	false	false
[enter_credential] - ((AuthenticationMethod && Anonymous && Forum) || (AuthenticationMethod && RegisteredUser && !Anonymous && Forum))	false	false	false
[choose_pseudo] - (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)	false	false	false
[choose_pseudo, send_msg] - (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)	false	false	false
[choose_pseudo, abort] - (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)	false	false	false
[choose_pseudo, log_in] - (AuthenticationMethod && Anonymous && Forum)	false	false	false
[choose_pseudo, log_in] - (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)	false	false	false
[choose_pseudo, choose_pseudo] - (AuthenticationMethod && Anonymous && Forum)	false	false	false
[choose_pseudo, choose_pseudo] - (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)	false	false	false
[choose_pseudo, enter_credential] - (AuthenticationMethod && Anonymous && Forum)	false	false	false
[choose_pseudo, enter_credential] - (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)	false	false	false
[choose_pseudo, send_msg, abort] - (AuthenticationMethod && Anonymous && Forum)	false	false	false
[choose_pseudo, send_msg, abort] - (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)	false	false	false
[choose_pseudo, send_msg, send_msg] - (AuthenticationMethod && Anonymous && Forum)	false	false	false
[choose_pseudo, send_msg, send_msg] - (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)	false	false	false
[choose_pseudo, send_msg, log_in] - (AuthenticationMethod && Anonymous && Forum)	false	false	false
[choose_pseudo, send_msg, log_in] - (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)	false	false	false
[choose_pseudo, send_msg, choose_pseudo] - (AuthenticationMethod && Anonymous && Forum)	false	false	false
[choose_pseudo, send_msg, choose_pseudo] - (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)	false	false	false
[choose_pseudo, send_msg, enter_credential] - (AuthenticationMethod && Anonymous && Forum)	false	false	false
[choose_pseudo, send_msg, enter_credential] - (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)	false	false	false
[choose_pseudo, abort, abort] - (AuthenticationMethod && Anonymous && Forum)	false	false	false
[choose_pseudo, abort, abort] - (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)	false	false	false
[choose_pseudo, abort, send_msg] - (AuthenticationMethod && Anonymous && Forum)	false	false	false
[choose_pseudo, abort, send_msg] - (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)	false	false	false
[choose_pseudo, abort, log_in] - (AuthenticationMethod && Anonymous && Forum)	false	false	false
[choose_pseudo, abort, log_in] - (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)	false	false	false
[choose_pseudo, abort, choose_pseudo] - (AuthenticationMethod && Anonymous && Forum)	false	false	false
[choose_pseudo, abort, choose_pseudo] - (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)	false	false	false
[choose_pseudo, abort, enter_credential] - (AuthenticationMethod && Anonymous && Forum)	false	false	false
[choose_pseudo, abort, enter_credential] - (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)	false	false	false

Figure A.3: Forum case study: observation table after the second learning round

```
<?xml version="1.0" encoding="UTF-8"?>
<fts>
    <start>Q_0</start>
    <states>
        <state id="Q_1">
            <transition target="Q_3" action="abort"
                fexpression="(AuthenticationMethod &amp;amp;&amp
                ; Anonymous &amp;amp;&amp; Forum)"></transition>
            <transition target="Q_2" action="send_msg"
                fexpression="(AuthenticationMethod &amp;amp;&amp
                ; Anonymous &amp;amp;&amp; Forum)"></transition>
        </state>
        <state id="Q_0">
            <transition target="Q_1" action="choose_pseudo"
                fexpression="(AuthenticationMethod &amp;amp;&amp
                ; Anonymous &amp;amp;&amp; Forum)"></transition>
        </state>
        <state id="Q_3"></state>
        <state id="Q_2"></state>
    </states>
</fts>
```

Listing A.3: Forum case study: FDFA in XML format after the second learning round

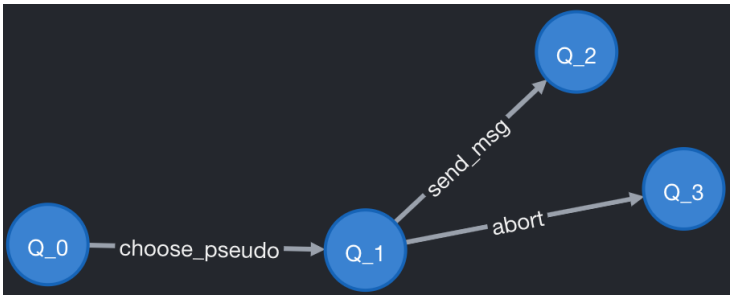

Figure A.4: Forum case study: FDFA visualisation after the second learning round

A.4 Artefacts Obtained After the Third Learning Round

After the third learning round, no more counterexamples can be found. Figure A.5 describes the final observation table, which can be simplified in the table from Figure A.6. The resulting FDFA is presented in Listing A.4 and Figure A.7.

table_2

OT	[c]	[abort]	[send_msg]	[log_in, abort]	[log_in, send_msg]	[choose_pseudo, abort]	[choose_pseudo, send_msg]
(0, 1) [c] – [(AuthenticationMethod && Anonymous && Forum) ! (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)]	false	false	false	false	false	(AuthenticationMethod && Cancelation && Anonymous && Forum)	(AuthenticationMethod && Anonymous && Forum)
(0, 1) [choose_pseudo] – [AuthenticationMethod && Anonymous && Forum]	false	(AuthenticationMethod && Cancelation && Anonymous && Forum)	(AuthenticationMethod && Anonymous && Forum)	false	false	false	false
(0, 2) [choose_pseudo, send_msg] – [AuthenticationMethod && Anonymous && Forum]	(AuthenticationMethod && Anonymous && Forum)	false	false	false	false	false	false
(0, 2) [choose_pseudo, abort] – [AuthenticationMethod && Anonymous && Forum]	(AuthenticationMethod && Cancellation && Anonymous && Forum)	false	false	false	false	false	false
(0, 4) [enter_credential] – [AuthenticationMethod && RegisteredUser && Forum]	false	false	(AuthenticationMethod && RegisteredUser && Cancellation && Forum)	(AuthenticationMethod && RegisteredUser && Cancellation && Forum)	(AuthenticationMethod && RegisteredUser && Forum)	false	false
(0, 5) [enter_credential, log_in] – [AuthenticationMethod && RegisteredUser && Forum]	false	(AuthenticationMethod && RegisteredUser && Cancellation && Forum)	(AuthenticationMethod && RegisteredUser && Forum)	false	false	false	false
(0, 6) [enter_credential, log_in, send_msg] – [AuthenticationMethod && RegisteredUser && Forum]	(AuthenticationMethod && RegisteredUser && Forum)	false	false	false	false	false	false
(0, 7) [enter_credential, log_in, abort] – [AuthenticationMethod && RegisteredUser && Cancellation && Forum]	(AuthenticationMethod && RegisteredUser && Cancellation && Forum)	false	false	false	false	false	false
[abort] – [(AuthenticationMethod && Anonymous && Forum) ! (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)]	false	false	false	false	false	false	false
[send_msg] – [(AuthenticationMethod && Anonymous && Forum) ! (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)]	false	false	false	false	false	false	false
[log_in] – [(AuthenticationMethod && Anonymous && Forum) ! (AuthenticationMethod && RegisteredUser && !Anonymous && Forum)]	false	false	false	false	false	false	false
[choose_pseudo] – [AuthenticationMethod && RegisteredUser && !Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, send_msg] – [AuthenticationMethod && RegisteredUser && !Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, abort] – [AuthenticationMethod && RegisteredUser && !Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, log_in] – [AuthenticationMethod && Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, log_in] – [AuthenticationMethod && RegisteredUser && !Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, choose_pseudo] – [AuthenticationMethod && Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, choose_pseudo] – [AuthenticationMethod && RegisteredUser && !Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, enter_credential] – [AuthenticationMethod && Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, enter_credential] – [AuthenticationMethod && RegisteredUser && !Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, send_msg, abort] – [AuthenticationMethod && Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, send_msg, abort] – [AuthenticationMethod && RegisteredUser && !Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, send_msg, send_msg] – [AuthenticationMethod && Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, send_msg, send_msg] – [AuthenticationMethod && RegisteredUser && !Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, send_msg, log_in] – [AuthenticationMethod && Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, send_msg, log_in] – [AuthenticationMethod && RegisteredUser && !Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, abort, choose_pseudo] – [AuthenticationMethod && Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, abort, choose_pseudo] – [AuthenticationMethod && RegisteredUser && !Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, abort, enter_credential] – [AuthenticationMethod && Anonymous && Forum]	false	false	false	false	false	false	false
[choose_pseudo, abort, enter_credential] – [AuthenticationMethod && RegisteredUser && !Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, send_msg] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, abort] – [AuthenticationMethod && RegisteredUser && Forum]	false	false	false	false	false	false	false
[enter_credential, abort] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, send_msg] – [AuthenticationMethod && RegisteredUser && Forum]	false	false	false	false	false	false	false
[enter_credential, send_msg] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, choose_pseudo] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, choose_pseudo] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, enter_credential] – [AuthenticationMethod && RegisteredUser && Forum]	false	false	false	false	false	false	false
[enter_credential, enter_credential] – [AuthenticationMethod && RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, abort] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, log_in] – [AuthenticationMethod && RegisteredUser && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, log_in] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, choose_pseudo] – [AuthenticationMethod && RegisteredUser && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, choose_pseudo] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, enter_credential] – [AuthenticationMethod && RegisteredUser && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, enter_credential] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, send_msg, abort] – [AuthenticationMethod && RegisteredUser && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, send_msg, abort] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, send_msg, send_msg] – [AuthenticationMethod && RegisteredUser && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, send_msg, send_msg] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, send_msg, log_in] – [AuthenticationMethod && RegisteredUser && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, send_msg, log_in] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, send_msg, choose_pseudo] – [AuthenticationMethod && RegisteredUser && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, send_msg, choose_pseudo] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, send_msg, enter_credential] – [AuthenticationMethod && RegisteredUser && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, send_msg, enter_credential] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, abort, abort] – [AuthenticationMethod && RegisteredUser && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, abort, abort] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, abort, send_msg] – [AuthenticationMethod && RegisteredUser && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, abort, send_msg] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, abort, log_in] – [AuthenticationMethod && RegisteredUser && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, abort, log_in] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, abort, choose_pseudo] – [AuthenticationMethod && RegisteredUser && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, abort, choose_pseudo] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, abort, enter_credential] – [AuthenticationMethod && RegisteredUser && Forum]	false	false	false	false	false	false	false
[enter_credential, log_in, abort, enter_credential] – [AuthenticationMethod && !RegisteredUser && Anonymous && Forum]	false	false	false	false	false	false	false

Figure A.5: Forum case study: observation table after algorithm completion

toy_ot_simplified

OT	[c]	[abort]	[send_msg]	[log_in, abort]	[log_in, send_msg]	[choose_pseudo, abort]	[choose_pseudo, send_msg]
(Q_0) [c] - ((AuthenticationMethod && Anonymous && Forum) || (AuthenticationMethod && RegisteredUser && !Anonymous && Forum))	false	false	false	false	false	(AuthenticationMethod && Cancellation && Anonymous && Forum)	(AuthenticationMethod && Anonymous && Forum)
(Q_1) [choose_pseudo] - (AuthenticationMethod && Anonymous && Forum)	false	(AuthenticationMethod && Cancellation && Anonymous && Forum)	(AuthenticationMethod && Anonymous && Forum)	false	false	false	false
(Q_2) [choose_pseudo, send_msg] - (AuthenticationMethod && Anonymous && Forum)	(AuthenticationMethod && Anonymous && Forum)	false	false	false	false	false	false
(Q_3) [choose_pseudo, abort] - (AuthenticationMethod && Anonymous && Forum)	(AuthenticationMethod && Cancellation && Anonymous && Forum)	false	false	false	false	false	false
(Q_4) [enter_credential] - (AuthenticationMethod && RegisteredUser && Forum)	false	false	false	(AuthenticationMethod && RegisteredUser && Cancellation && Forum)	(AuthenticationMethod && RegisteredUser && Forum)	false	false
(Q_5) [enter_credential, log_in] - (AuthenticationMethod && RegisteredUser && Forum)	false	(AuthenticationMethod && RegisteredUser && Cancellation && Forum)	(AuthenticationMethod && RegisteredUser && Forum)	false	false	false	false
(Q_6) [enter_credential, log_in, send_msg] - (AuthenticationMethod && RegisteredUser && Forum)	(AuthenticationMethod && RegisteredUser && Forum)	false	false	false	false	false	false
(Q_7) [enter_credential, log_in, abort] - (AuthenticationMethod && RegisteredUser && Forum)	(AuthenticationMethod && RegisteredUser && Cancellation && Forum)	false	false	false	false	false	false

Figure A.6: Forum case study: simplified observation table

```
<?xml version="1.0" encoding="UTF-8"?>
<fts>
    <start>Q_0</start>
    <states>
        <state id="Q_1">
            <transition target="Q_3" action="abort" fexpression="
                (AuthenticationMethod &amp;&amp; Anonymous &amp
                ;&amp; Forum)"></transition>
            <transition target="Q_2" action="send_msg"
                fexpression="(AuthenticationMethod &amp;&amp;
                Anonymous &amp;&amp; Forum)"></transition>
        </state>
        <state id="Q_0">
            <transition target="Q_1" action="choose_pseudo"
                fexpression="(AuthenticationMethod &amp;&amp;
                Anonymous &amp;&amp; Forum)"></transition>
            <transition target="Q_4" action="enter_credential"
                fexpression="(AuthenticationMethod &amp;&amp;
                RegisteredUser &amp;&amp; Forum)"></transition>
        </state>
        <state id="Q_3"></state>
        <state id="Q_2"></state>
        <state id="Q_5">
            <transition target="Q_7" action="abort" fexpression="
                (AuthenticationMethod &amp;&amp; RegisteredUser
                &amp;&amp; Forum)"></transition>
            <transition target="Q_6" action="send_msg"
                fexpression="(AuthenticationMethod &amp;&amp;
                RegisteredUser &amp;&amp; Forum)"></transition>
        </state>
        <state id="Q_4">
            <transition target="Q_5" action="log_in" fexpression=
                "(AuthenticationMethod &amp;&amp; RegisteredUser
                 &amp;&amp; Forum)"></transition>
        </state>
        <state id="Q_7"></state>
        <state id="Q_6"></state>
    </states>
</fts>
```

Listing A.4: Forum case study: FDFA in XML format after algorithm completion

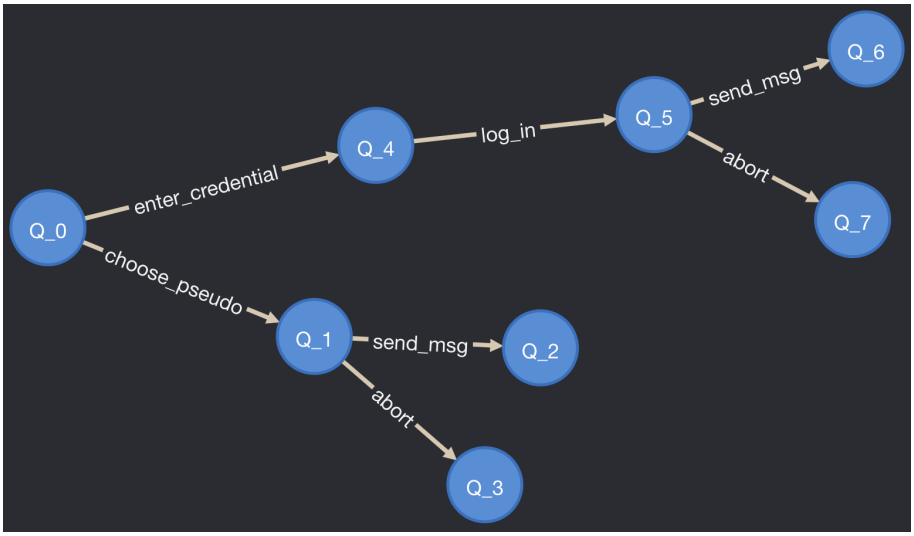

Figure A.7: Forum case study: FDFA visualisation after algorithm completion

APPENDIX B

EXAMPLE OF FDFA SIMPLIFICATION

Figure B.1 presents the FDFA obtained after learning the soda vending machine case study (see Section 7.2.2) but before applying the simplification algorithm. Figure B.2 presents the same FDFA after this simplification.

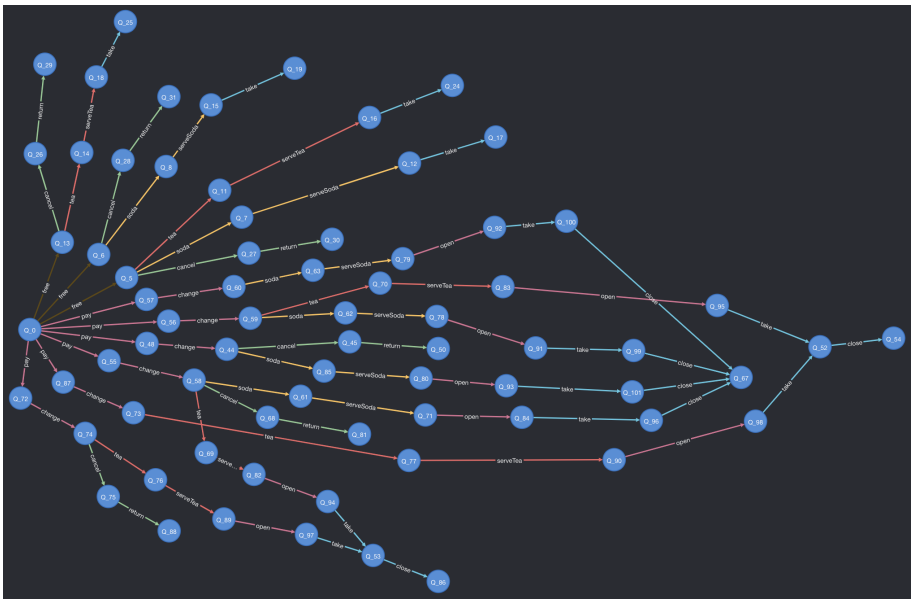

Figure B.1: Learned FDFA of the Soda Vending Machine without simplification

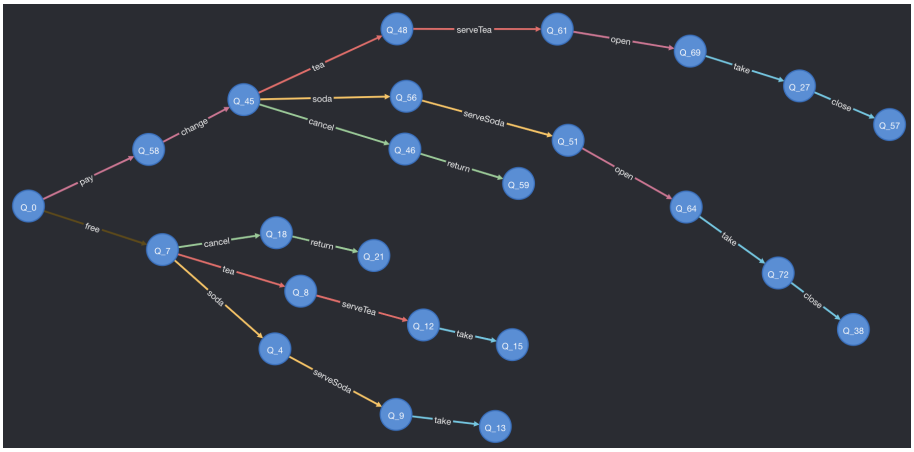

Figure B.2: Simplified FDFA of the Soda Vending Machine

APPENDIX C

VARYMINIONS METRICS

This appendix contains 6 tables (one per dataset) representing the average and standard deviation for four metrics computed on 10 iterations. Accuracy, precision, recall and F1-score were computed based on definitions provided in Section 11.5.

The first three columns of the tables show the hyperparameter values for each of the RNNs' parameterisations. For conciseness, we do not report in these tables hyperparameters that were fixed to a single value, such as the batch size or the number of epochs. Indeed, we discussed them in Section 11.3. The other columns report the average and standard deviation of accuracy, precision, recall and F1-score.

C.1 BPIC15

Table C.1: Results for dataset BPIC15: Averaged and standard deviations of different metrics over 10 runs. Each line corresponds to a parameterisation of a RNN.

Dataset	Model	Loss	Activation	Accuracy		Precision		Recall		F1Score	
				Avg	Sd	Avg	Sd	Avg	Sd	Avg	Sd
BPIC15	LSTM	mse	tanh	0.8848	0.0074	0.8854	0.0074	0.8848	0.0074	0.8845	0.0078
BPIC15	LSTM	bin_ce	sigmoid	0.8806	0.0062	0.8818	0.006	0.8806	0.0062	0.8804	0.0064
BPIC15	LSTM	mse	sigmoid	0.8783	0.0069	0.8796	0.0064	0.8783	0.0069	0.8782	0.0068
BPIC15	GRU	mse	tanh	0.8741	0.007	0.8747	0.0071	0.8741	0.007	0.8733	0.0075
BPIC15	GRU	bin_ce-logits	sigmoid	0.8733	0.0097	0.8738	0.0103	0.8733	0.0097	0.8728	0.0101
BPIC15	GRU	bin_ce	sigmoid	0.8677	0.0126	0.8683	0.0117	0.8677	0.0126	0.8672	0.0125
BPIC15	GRU	mse	sigmoid	0.8623	0.0077	0.863	0.0081	0.8623	0.0077	0.8619	0.0079
BPIC15	LSTM	bin_ce-logits	sigmoid	0.8571	0.0095	0.8597	0.0095	0.8571	0.0095	0.8566	0.0099
BPIC15	LSTM	manhattan	tanh	0.8257	0.0145	0.8311	0.0155	0.8257	0.0145	0.8228	0.0147
BPIC15	GRU	bin_ce-logits	tanh	0.7937	0.0596	0.7502	0.1143	0.7937	0.0596	0.7676	0.0902
BPIC15	GRU	bin_ce	tanh	0.7898	0.021	0.7889	0.0206	0.7898	0.021	0.7841	0.0224
BPIC15	GRU	manhattan	tanh	0.7858	0.0092	0.7896	0.0109	0.7858	0.0092	0.7739	0.0105
BPIC15	LSTM	jaccard	sigmoid	0.7858	0.0447	0.8072	0.0581	0.7858	0.0447	0.7541	0.0666
BPIC15	LSTM	bin_ce	tanh	0.7724	0.0778	0.7864	0.0625	0.7724	0.0778	0.7666	0.0839
BPIC15	LSTM	bin_ce-logits	tanh	0.7436	0.1165	0.7129	0.1585	0.7436	0.1165	0.6944	0.1604
BPIC15	GRU	jaccard	sigmoid	0.6962	0.0583	0.706	0.0696	0.6962	0.0583	0.6566	0.0861
BPIC15	LSTM	jaccard	tanh	0.6098	0.0438	0.588	0.0276	0.6098	0.0438	0.5416	0.0565
BPIC15	GRU	jaccard	tanh	0.5529	0.0244	0.5832	0.0511	0.5529	0.0244	0.4847	0.042
BPIC15	GRU	manhattan	sigmoid	0.2538	0.0585	0.1223	0.0912	0.2538	0.0585	0.1194	0.0627
BPIC15	LSTM	manhattan	sigmoid	0.2313	0.0552	0.0893	0.0776	0.2313	0.0552	0.0926	0.0505

C.2 BPIC20

Table C.2: Results for dataset BPIC20: Averaged and standard deviations of different metrics over 10 runs. Each line corresponds to a parameterisation of a RNN.

Dataset	Model	Loss	Activation	Accuracy		Precision		Recall		F1Score	
				Avg	Sd	Avg	Sd	Avg	Sd	Avg	Sd
BPIC20	LSTM	mse	sigmoid	0.8744	0.0397	0.8876	0.0279	0.8744	0.0397	0.8716	0.0424
BPIC20	LSTM	mse	tanh	0.8585	0.0375	0.8764	0.0261	0.8585	0.0375	0.8539	0.0416
BPIC20	LSTM	bin_ce-logits	sigmoid	0.8541	0.0714	0.8736	0.0499	0.8541	0.0714	0.8425	0.0852
BPIC20	LSTM	bin_ce	sigmoid	0.8397	0.0642	0.8598	0.0485	0.8397	0.0642	0.8308	0.0744
BPIC20	GRU	mse	sigmoid	0.8258	0.0897	0.829	0.0997	0.8258	0.0897	0.8148	0.1086
BPIC20	GRU	mse	tanh	0.8198	0.0489	0.8508	0.0287	0.8198	0.0489	0.8094	0.0574
BPIC20	GRU	bin_ce-logits	sigmoid	0.7895	0.0617	0.8229	0.0499	0.7895	0.0617	0.7722	0.0728
BPIC20	GRU	bin_ce	sigmoid	0.7758	0.0428	0.81	0.0338	0.7758	0.0428	0.758	0.054
BPIC20	LSTM	bin_ce-logits	tanh	0.7569	0.072	0.7797	0.0806	0.7569	0.072	0.7284	0.0924
BPIC20	GRU	bin_ce-logits	tanh	0.7303	0.1003	0.7828	0.0714	0.7303	0.1003	0.6852	0.1422
BPIC20	LSTM	bin_ce	tanh	0.6469	0.126	0.6276	0.2199	0.6469	0.126	0.5695	0.1876
BPIC20	GRU	bin_ce	tanh	0.5918	0.041	0.5966	0.205	0.5918	0.041	0.478	0.0642
BPIC20	LSTM	manhattan	tanh	0.5878	0.0332	0.5173	0.1275	0.5878	0.0332	0.4707	0.0459
BPIC20	GRU	manhattan	tanh	0.5826	0.0458	0.6858	0.0653	0.5826	0.0458	0.4506	0.0629
BPIC20	GRU	jaccard	sigmoid	0.5449	0.0139	0.3075	0.0176	0.5449	0.0139	0.3898	0.0165
BPIC20	LSTM	jaccard	tanh	0.4661	0.0236	0.2321	0.0435	0.4661	0.0236	0.2995	0.0278
BPIC20	LSTM	manhattan	sigmoid	0.4629	0.0115	0.2144	0.0107	0.4629	0.0115	0.293	0.0123
BPIC20	GRU	jaccard	tanh	0.4602	0.0229	0.2362	0.0367	0.4602	0.0229	0.2933	0.0242
BPIC20	GRU	manhattan	sigmoid	0.4584	0.0217	0.2105	0.02	0.4584	0.0217	0.2884	0.023
BPIC20	LSTM	jaccard	sigmoid	0.4576	0.0144	0.2096	0.0132	0.4576	0.0144	0.2875	0.0152

C.3 Claroline Dissimilar 10

Table C.3: Results for dataset Claroline Dissimilar 10: Averaged and standard deviations of different metrics over 10 runs. Each line corresponds to a parameterisation of a RNN.

Dataset	Model	Loss	Activation	Accuracy		Precision		Recall		F1Score	
				Avg	Sd	Avg	Sd	Avg	Sd	Avg	Sd
claroline-dis_10	GRU	bin_ce	sigmoid	0.9968	0.0006	0.9968	0.0006	0.9968	0.0006	0.9968	0.0006
claroline-dis_10	GRU	bin_ce-logits	sigmoid	0.9964	0.0003	0.9965	0.0003	0.9964	0.0003	0.9964	0.0003
claroline-dis_10	GRU	mse	sigmoid	0.9064	0.1868	0.8842	0.2111	0.9064	0.1868	0.8905	0.2059
claroline-dis_10	LSTM	mse	tanh	0.8536	0.2532	0.8181	0.3077	0.8536	0.2532	0.8245	0.2977
claroline-dis_10	LSTM	mse	sigmoid	0.7458	0.2554	0.6726	0.3102	0.7458	0.2554	0.6913	0.2986
claroline-dis_10	LSTM	bin_ce-logits	sigmoid	0.6885	0.217	0.6007	0.2605	0.6885	0.217	0.6147	0.2531
claroline-dis_10	LSTM	bin_ce	sigmoid	0.6871	0.3138	0.6213	0.3627	0.6871	0.3138	0.6258	0.362
claroline-dis_10	LSTM	bin_ce	tanh	0.3592	0.2734	0.2388	0.27	0.3592	0.2734	0.2636	0.2772
claroline-dis_10	LSTM	manhattan	tanh	0.2687	0.0785	0.1686	0.0583	0.2687	0.0785	0.1767	0.0732
claroline-dis_10	GRU	bin_ce	tanh	0.2599	0.3392	0.1774	0.3597	0.2599	0.3392	0.1875	0.3619
claroline-dis_10	GRU	jaccard	tanh	0.1998	0.1043	0.0595	0.0644	0.1998	0.1043	0.0843	0.079
claroline-dis_10	LSTM	jaccard	tanh	0.1796	0.0786	0.0552	0.0564	0.1796	0.0786	0.0728	0.0643
claroline-dis_10	GRU	manhattan	sigmoid	0.1012	0.0015	0.0103	0.0003	0.1012	0.0015	0.0186	0.0005
claroline-dis_10	GRU	bin_ce-logits	tanh	0.1006	0.0012	0.0101	0.0002	0.1006	0.0012	0.0184	0.0004
claroline-dis_10	GRU	mse	tanh	0.0998	0.0017	0.01	0.0003	0.0998	0.0017	0.0181	0.0006
claroline-dis_10	LSTM	bin_ce-logits	tanh	0.0997	0.0019	0.0116	0.005	0.0997	0.0019	0.0181	0.0007
claroline-dis_10	LSTM	manhattan	sigmoid	0.0996	0.0016	0.0099	0.0003	0.0996	0.0016	0.018	0.0006
claroline-dis_10	LSTM	jaccard	sigmoid	0.0995	0.0016	0.0099	0.0003	0.0995	0.0016	0.018	0.0006
claroline-dis_10	GRU	manhattan	tanh	0.0991	0.0022	0.0098	0.0004	0.0991	0.0022	0.0179	0.0007
claroline-dis_10	GRU	jaccard	sigmoid	0.0989	0.0022	0.0098	0.0004	0.0989	0.0022	0.0178	0.0008

C.4 Claroline Random 10

Table C.4: Results for dataset Claroline Random 10: Averaged and standard deviations of different metrics over 10 runs. Each line corresponds to a parameterisation of a RNN.

Dataset	Model	Loss	Activation	Accuracy		Precision		Recall		F1Score	
				Avg	Sd	Avg	Sd	Avg	Sd	Avg	Sd
claroline-rand_10	GRU	bin_ce	sigmoid	0.9861	0.0321	0.9811	0.0482	0.9861	0.0321	0.9828	0.0428
claroline-rand_10	GRU	bin_ce-logits	sigmoid	0.9664	0.0947	0.9548	0.1317	0.9664	0.0947	0.9581	0.1211
claroline-rand_10	GRU	mse	sigmoid	0.9154	0.1063	0.8869	0.1355	0.9154	0.1063	0.8945	0.1292
claroline-rand_10	LSTM	mse	tanh	0.8886	0.1639	0.859	0.2065	0.8886	0.1639	0.8619	0.1999
claroline-rand_10	LSTM	bin_ce	sigmoid	0.7678	0.2389	0.7068	0.2775	0.7678	0.2389	0.7211	0.2713
claroline-rand_10	LSTM	mse	sigmoid	0.7352	0.2056	0.6751	0.2397	0.7352	0.2056	0.6863	0.2358
claroline-rand_10	LSTM	bin_ce-logits	sigmoid	0.7183	0.1685	0.6454	0.2028	0.7183	0.1685	0.6573	0.2015
claroline-rand_10	LSTM	bin_ce	tanh	0.4748	0.1905	0.3559	0.2044	0.4748	0.1905	0.382	0.2052
claroline-rand_10	GRU	jaccard	tanh	0.2099	0.2221	0.1228	0.2179	0.2099	0.2221	0.1342	0.2249
claroline-rand_10	LSTM	manhattan	tanh	0.1902	0.0734	0.1045	0.0698	0.1902	0.0734	0.1076	0.0661
claroline-rand_10	GRU	bin_ce	tanh	0.1793	0.2517	0.0928	0.2621	0.1793	0.2517	0.1023	0.2661
claroline-rand_10	LSTM	jaccard	tanh	0.1385	0.0696	0.0271	0.0311	0.1385	0.0696	0.0426	0.0441
claroline-rand_10	LSTM	manhattan	sigmoid	0.1012	0.0017	0.0102	0.0003	0.1012	0.0017	0.0186	0.0006
claroline-rand_10	GRU	manhattan	tanh	0.1007	0.0015	0.0101	0.0003	0.1007	0.0015	0.0184	0.0005
claroline-rand_10	GRU	bin_ce-logits	tanh	0.1007	0.0013	0.0101	0.0003	0.1007	0.0013	0.0184	0.0005
claroline-rand_10	LSTM	bin_ce-logits	tanh	0.1004	0.0019	0.0101	0.0004	0.1004	0.0019	0.0183	0.0007
claroline-rand_10	GRU	mse	tanh	0.1001	0.0018	0.01	0.0004	0.1001	0.0018	0.0182	0.0006
claroline-rand_10	GRU	manhattan	sigmoid	0.0991	0.0018	0.0098	0.0004	0.0991	0.0018	0.0179	0.0006
claroline-rand_10	LSTM	jaccard	sigmoid	0.0989	0.0016	0.0098	0.0003	0.0989	0.0016	0.0178	0.0005
claroline-rand_10	GRU	jaccard	sigmoid	0.0983	0.0014	0.0097	0.0003	0.0983	0.0014	0.0176	0.0005

C.5 Claroline Dissimilar 50

Table C.5: Results for dataset Claroline Dissimilar 50: Averaged and standard deviations of different metrics over 10 runs. Each line corresponds to a parameterisation of a RNN.

Dataset	Model	Loss	Activation	Accuracy		Precision		Recall		F1Score	
				Avg	Sd	Avg	Sd	Avg	Sd	Avg	Sd
claroline-dis_50	LSTM	bin_ce	sigmoid	0.8001	0.1387	0.7631	0.1562	0.8001	0.1387	0.7603	0.1532
claroline-dis_50	LSTM	bin_ce-logits	sigmoid	0.7833	0.2161	0.7367	0.2528	0.7833	0.2161	0.7403	0.25
claroline-dis_50	GRU	bin_ce-logits	sigmoid	0.7225	0.381	0.6943	0.3834	0.7225	0.381	0.7002	0.3848
claroline-dis_50	LSTM	mse	tanh	0.6211	0.0379	0.5221	0.045	0.6211	0.0379	0.5373	0.0374
claroline-dis_50	GRU	bin_ce	sigmoid	0.5256	0.4459	0.4901	0.4408	0.5256	0.4459	0.4975	0.4442
claroline-dis_50	LSTM	mse	sigmoid	0.2437	0.25	0.197	0.223	0.2437	0.25	0.1983	0.2226
claroline-dis_50	GRU	mse	sigmoid	0.2089	0.2153	0.1375	0.1617	0.2089	0.2153	0.1507	0.1747
claroline-dis_50	GRU	bin_ce	tanh	0.1015	0.2253	0.0761	0.2231	0.1015	0.2253	0.0781	0.2257
claroline-dis_50	GRU	jaccard	tanh	0.0778	0.0386	0.0209	0.0217	0.0778	0.0386	0.026	0.0251
claroline-dis_50	LSTM	jaccard	tanh	0.0389	0.0131	0.0019	0.0011	0.0389	0.0131	0.0036	0.002
claroline-dis_50	LSTM	jaccard	sigmoid	0.0273	0.01	0.0013	0.0016	0.0273	0.01	0.0024	0.0026
claroline-dis_50	GRU	manhattan	sigmoid	0.0201	0.0003	0.0004	0.0	0.0201	0.0003	0.0008	0.0
claroline-dis_50	GRU	manhattan	tanh	0.0201	0.0008	0.0004	0.0001	0.0201	0.0008	0.0009	0.0002
claroline-dis_50	LSTM	bin_ce-logits	tanh	0.0201	0.0004	0.0004	0.0	0.0201	0.0004	0.0008	0.0
claroline-dis_50	LSTM	manhattan	sigmoid	0.0201	0.0006	0.0004	0.0	0.0201	0.0006	0.0008	0.0
claroline-dis_50	GRU	mse	tanh	0.02	0.0004	0.0004	0.0	0.02	0.0004	0.0008	0.0
claroline-dis_50	GRU	bin_ce-logits	tanh	0.02	0.0004	0.0004	0.0	0.02	0.0004	0.0008	0.0
claroline-dis_50	LSTM	manhattan	tanh	0.0199	0.0004	0.0004	0.0	0.0199	0.0004	0.0008	0.0
claroline-dis_50	GRU	jaccard	sigmoid	0.0192	0.0002	0.0004	0.0	0.0192	0.0002	0.0007	0.0
claroline-dis_50	LSTM	bin_ce	tanh	0.0158	0.0078	0.001	0.0009	0.0158	0.0078	0.0014	0.0012

C.6 Claroline Random 50

Table C.6: Results for dataset Claroline Random 50: Averaged and standard deviations of different metrics over 10 runs. Each line corresponds to a parameterisation of a RNN.

Dataset	Model	Loss	Activation	Accuracy		Precision		Recall		F1Score	
				Avg	Sd	Avg	Sd	Avg	Sd	Avg	Sd
claroline-rand_50	GRU	bin_ce	sigmoid	0.9562	0.0785	0.9442	0.1023	0.9562	0.0785	0.9475	0.0953
claroline-rand_50	GRU	bin_ce-logits	sigmoid	0.9498	0.1013	0.9368	0.1304	0.9498	0.1013	0.939	0.1238
claroline-rand_50	LSTM	bin_ce-logits	sigmoid	0.9197	0.1032	0.9085	0.1109	0.9197	0.1032	0.9041	0.1209
claroline-rand_50	LSTM	bin_ce	sigmoid	0.8855	0.0936	0.8607	0.1175	0.8855	0.0936	0.8631	0.1106
claroline-rand_50	LSTM	mse	sigmoid	0.698	0.221	0.6235	0.248	0.698	0.221	0.6429	0.2421
claroline-rand_50	LSTM	mse	tanh	0.6136	0.0387	0.5077	0.046	0.6136	0.0387	0.5289	0.0372
claroline-rand_50	GRU	mse	sigmoid	0.5852	0.3281	0.5186	0.322	0.5852	0.3281	0.5331	0.324
claroline-rand_50	GRU	mse	tanh	0.4372	0.288	0.3482	0.2402	0.4372	0.288	0.3691	0.2544
claroline-rand_50	GRU	bin_ce	tanh	0.3667	0.1106	0.3056	0.1063	0.3667	0.1106	0.3179	0.1041
claroline-rand_50	GRU	jaccard	sigmoid	0.2623	0.2071	0.1773	0.186	0.2623	0.2071	0.1923	0.1943
claroline-rand_50	LSTM	jaccard	sigmoid	0.1028	0.0811	0.0373	0.0467	0.1028	0.0811	0.0464	0.0546
claroline-rand_50	GRU	jaccard	tanh	0.0981	0.0224	0.0305	0.0166	0.0981	0.0224	0.0375	0.0185
claroline-rand_50	LSTM	jaccard	tanh	0.0745	0.0282	0.0135	0.0215	0.0745	0.0282	0.0186	0.0227
claroline-rand_50	LSTM	bin_ce	tanh	0.0395	0.0262	0.0205	0.0247	0.0395	0.0262	0.0195	0.0258
claroline-rand_50	GRU	manhattan	tanh	0.0394	0.0094	0.0232	0.0146	0.0394	0.0094	0.0205	0.0095
claroline-rand_50	LSTM	manhattan	tanh	0.0202	0.0004	0.0004	0.0	0.0202	0.0004	0.0008	0.0
claroline-rand_50	LSTM	manhattan	sigmoid	0.0199	0.0003	0.0004	0.0	0.0199	0.0003	0.0008	0.0
claroline-rand_50	GRU	bin_ce-logits	tanh	0.0198	0.0004	0.0007	0.0007	0.0198	0.0004	0.0008	0.0
claroline-rand_50	LSTM	bin_ce-logits	tanh	0.0198	0.0005	0.0004	0.0	0.0198	0.0005	0.0008	0.0
claroline-rand_50	GRU	manhattan	sigmoid	0.0197	0.0006	0.0004	0.0	0.0197	0.0006	0.0008	0.0

me

BIBLIOGRAPHY

[1] Fides Aarts, Paul Fiterau-Brostean, Harco Kuppens, and Frits Vaandrager. Learning register automata with fresh value generation. In **Theoretical Aspects of Computing-ICTAC 2015: 12th International Colloquium, Cali, Colombia, October 29-31, 2015, Proceedings 12**, pages 165–183. Springer, 2015.

[2] Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits Vaandrager. Automata learning through counterexample guided abstraction refinement. In **FM 2012: Formal Methods: 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings 18**, pages 10–27. Springer, 2012.

[3] Mohamed Abdelrazek, John Grundy, and Amani Ibrahim. Towards Self-securing Software Systems: Variability Spectrum. In **Software Engineering for Variability Intensive Systems**, pages 119–130. Auerbach Publications, 2019.

[4] Mathieu Acher, Benoit Baudry, Patrick Heymans, Anthony Cleve, and Jean-Luc Hainaut. Support for reverse engineering and maintaining feature models. In Stefania Gnesi, Philippe Collet, and Klaus Schmid, editors, **VaMoS**, page 20. ACM, 2013.

[5] Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. On extracting feature models from product descriptions. In **Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems**, pages 45–54, 2012.

[6] Hasan Ibne Akram, Colin La Higuera, and Claudia Eckert. Actively learning probabilistic subsequential transducers. In **International Conference on Grammatical Inference**, pages 19–33, 2012.

[7] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, and Jean-Marc Jézéquel. Sampling Effect on Performance Prediction of Configurable Systems: A Case Study. In **Proceedings of the ACM/SPEC International Conference on Performance Engineering**, pages 277–288. ACM, 2020.

[8] Benoit Amand, Maxime Cordy, Patrick Heymans, Mathieu Acher, Paul Temple, and Jean-Marc Jézéquel. Towards learning-aided configuration in 3D printing: Feasibility study and application to defect prediction. In **Proceedings of the**

13th International Workshop on Variability Modelling of Software-Intensive Systems, pages 1–9. ACM, 2019.

[9] Dana Angluin. Learning regular sets from queries and counterexamples. **Information and computation**, 75(2):87–106, 1987.

[10] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. **Feature-Oriented Software Product Lines - Concepts and Implementation**. Springer, 2013.

[11] Sven Apel, Alexander Von Rhein, Thomas Thüm, and Christian Kästner. Feature-interaction detection based on feature-based specifications. **Computer Networks**, 57(12):2399–2409, 2013.

[12] Eleonora Arganese, Alessandro Fantechi, Stefania Gnesi, and Laura Semini. Nuts and Bolts of Extracting Variability Models from Natural Language Requirements Documents. In **Integrating Research and Practice in Software Engineering**, pages 125–143. Springer, 2020.

[13] Patrizia Asirelli, Maurice H Ter Beek, Stefania Gnesi, and Alessandro Fantechi. Formal description of variability in product families. In **2011 15th International Software Product Line Conference**, pages 130–139. IEEE, 2011.

[14] Wesley KG Assunção, Silvia R Vergilio, and Roberto E Lopez-Herrejon. Automatic extraction of product line architecture and feature models from UML class diagram variants. **Information and Software Technology**, 117:106198, 2020.

[15] Wesley Klewerton Guez Assunção, Roberto Erick Lopez-Herrejon, Lukas Linsbauer, Silvia Regina Vergilio, and Alexander Egyed. Reengineering legacy applications into software product lines: a systematic mapping. **Empirical Software Engineering**, 22:2972–3016, 2017.

[16] Nour Assy, Nguyen Ngoc Chan, and Walid Gaaloul. An automated approach for assisting the design of configurable process models. **IEEE transactions on services computing**, 8(6):874–888, 2015.

[17] Joanne M Atlee, Uli Fahrenberg, and Axel Legay. Measuring behaviour interactions between product-line features. In **2015 IEEE/ACM 3rd FME Workshop on Formal Methods in Software Engineering**, pages 20–25. IEEE, 2015.

[18] Thomas H Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In **Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages**, pages 165–178, 2012.

[19] Davide Bacciu, Stefania Gnesi, and Laura Semini. Using a Machine Learning Approach to Implement and Evaluate Product Line Features. In Maurice H. ter Beek and Alberto Lluch-Lafuente, editors, **Proceedings 11th International Workshop on Automated Specification and Verification of Web Systems,**

WWV 2015, Oslo, Norway, 23rd June 2015, volume 188 of **EPTCS**, pages 75–83. EPTCS, 2015.

[20] Christel Baier and Joost-Pieter Katoen. **Principles of model checking**. MIT press, 2008.

[21] Davide Basile. Applying supervisory control synthesis to priced featured automata and energy problems. **International Journal on Software Tools for Technology Transfer**, 21:679–689, 2019.

[22] Maurice H. ter Beek, Klaus Schmid, and Holger Eichelberger. Textual Variability Modeling Languages: An Overview and Considerations. In **Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B**, SPLC '19, page 151–157, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Tessa Belder, Maurice H ter Beek, and Erik P de Vink. Coherent branching feature bisimulation. **arXiv preprint arXiv:1504.03474**, 2015.

[24] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 years later: A Literature Review. **Information Systems**, 35(6):615 – 636, 2010.

[25] Nelly Bencomo, Peter Sawyer, Gordon S Blair, and Paul Grace. Dynamically adaptive systems are product lines too: using model-driven techniques to capture dynamic variability of adaptive systems. In **SPLC (2)**, pages 23–32, 2008.

[26] Harsh Beohar, Mahsa Varshosaz, and Mohammad Reza Mousavi. Basic behavioral models for software product lines: Expressiveness and testing pre-orders. **Science of Computer Programming**, 123:42–60, 2016.

[27] Szymon Bobek, Mateusz Baran, Krzysztof Kluza, and Grzegorz J Nalepa. Application of Bayesian Networks to Recommendations in Business Process Modeling. In **AIBP at AI* IA**, pages 41–50. Springer, 2013.

[28] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. Angluin-Style Learning of NFA. In **IJCAI**, volume 9, pages 1004–1009, 2009.

[29] Michael Borkowski, Walid Fdhila, Matteo Nardelli, Stefanie Rinderle-Ma, and Stefan Schulte. Event-based failure prediction in distributed business processes. **Information Systems**, 81:220–235, 2019.

[30] Mohamed Boussaa, Olivier Barais, Benoit Baudry, and Gerson Sunyé. Automatic non-functional testing of code generators families. **ACM SIGPLAN Notices**, 52(3):202–212, 2016.

[31] Zahra Dasht Bozorgi, Irene Teinemaa, Marlon Dumas, Marcello La Rosa, and Artem Polyvyanyy. Process mining meets causal machine learning: Discovering causal rules from event logs. In **2020 2nd International Conference on Process Mining (ICPM)**, pages 129–136. IEEE, 2020.

[32] Hong-Nhung Bui, Trong-Sinh Vu, Hien-Hanh Nguyen, Tri-Thanh Nguyen, and Quang-Thuy Ha. Exploiting CBOW and LSTM models to generate trace representation for process mining. In **Asian Conference on Intelligent Information and Database Systems**, pages 35–46. Springer, 2020.

[33] Hong-Nhung Bui, Trong-Sinh Vu, Tri-Thanh Nguyen, Thi-Cham Nguyen, and Quang-Thuy Ha. A compact trace representation using deep neural networks for process mining. In **2019 11th International Conference on Knowledge and Systems Engineering (KSE)**, pages 1–5. IEEE, 2019.

[34] Joos CAM Buijs, Boudewijn F van Dongen, and Wil MP van der Aalst. Mining configurable process models from collections of event logs. In **Business process management**, pages 33–48. Springer, 2013.

[35] Jeanderson Cândido, Maurício Aniche, and Arie van Deursen. Log-based software monitoring: a systematic mapping study. **PeerJ Computer Science**, 7:e489, 2021.

[36] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés, and Mike Hinchey. An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. **Journal of Systems and Software**, 91:3–23, 2014.

[37] Nicolás Cardozo and Ivana Dusparic. Learning run-time compositions of interacting adaptations. In **Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems**, pages 108–114, 2020.

[38] Sofia Cassel, Falk Howar, and Bengt Jonsson. RALib: A LearnLib extension for inferring EFSMs. **DIFTS. hp://www. faculty. ece. vt. edu/chaowang/dis2015/papers/paper**, 5, 2015.

[39] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Active learning for extended finite state machines. **Formal Aspects of Computing**, 28(2):233–263, 2016.

[40] Olivier Chapelle, Bernhard Schölkopf, and Alexander Zien. **Semi-Supervised Learning**. MIT press Cambridge, 2006.

[41] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In **Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)**, pages 1724–1734. Association for Computational Linguistics, 2014.

[42] François Chollet et al. Keras. https://keras.io, 2015.

[43] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In **NIPS 2014 Workshop on Deep Learning, December 2014**, 2014.

[44] Andreas Classen. Modelling with FTS: a Collection of Illustrative Examples. Technical Report P-CS-TR SPLMC-00000001, PReCISE Research Center, University of Namur, Namur, Belgium, 2010.

[45] Andreas Classen, Quentin Boucher, and Patrick Heymans. A Text-based Approach to Feature Modelling: Syntax and Semantics of TVL. **Science of Computer Programming, Special Issue on Software Evolution**, 2010.

[46] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. **IEEE Trans. Software Eng.**, 39(8):1069–1089, 2013.

[47] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In **Proceedings of the 33rd International Conference on Software Engineering**, pages 321–330, 2011.

[48] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In **Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1**, pages 335–344, 2010.

[49] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. Interaction testing of highly-configurable systems in the presence of constraints. In **Proceedings of the 2007 international symposium on Software testing and analysis**, pages 129–139, 2007.

[50] Maxime Cordy, Andreas Classen, Gilles Perrouin, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Simulation-based abstractions for software product-line model checking. In **2012 34th International Conference on Software Engineering (ICSE)**, pages 672–682. IEEE, 2012.

[51] Maxime Cordy, Xavier Devroey, Axel Legay, Gilles Perrouin, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Jean-François Raskin. A decade of featured transition systems. In **From Software Engineering to Formal Methods and Tools, and Back**, pages 285–312. Springer, 2019.

[52] Daniel Cruz, Eduardo Figueiredo, and Jabier Martinez. A literature review and comparison of three feature location techniques using ArgoUML-SPL. In **Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems**, pages 1–10, 2019.

[53] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. **Software Process Improvement and Practice**, 10(1):7–29, 2005.

[54] Carlos Diego N Damasceno, Mohammad Reza Mousavi, and Adenilso da Silva Simao. Learning to reuse: Adaptive model learning for evolving systems. In **International Conference on Integrated Formal Methods**, pages 138–156. Springer, 2019.

[55] Carlos Diego Nascimento Damasceno. LearningFFSM: Benchmark_SPL. https://github.com/damascenodiego/learningFFSM/tree/master/FFSM_diff/Benchmark_SPL, 2020. Accessed: 2023-05-10.

[56] Carlos Diego Nascimento Damasceno, Mohammad Reza Mousavi, and Adenilso da Silva Simao. Learning by sampling: learning behavioral family models from software product lines. **Empirical Software Engineering**, 26(1):1–46, 2021.

[57] Hugo Sica de Andrade, Eduardo Almeida, and Ivica Crnkovic. Architectural Bad Smells in Software Product Lines: An Exploratory Study. In **Proceedings of the WICSA 2014 Companion Volume**, WICSA '14 Companion, pages 1–6, New York, NY, USA, 2014. Association for Computing Machinery.

[58] Jochen De Weerdt, Seppe KLM vanden Broucke, Jan Vanthienen, and Bart Baesens. Leveraging process discovery with trace clustering and text mining for intelligent analysis of incident management processes. In **IEEE Congress on Evolutionary Computation**, pages 1–8. IEEE, 2012.

[59] TensorFlow Developers. TensorFlow. https://doi.org/10.5281/zenodo.4758419, May 2021.

[60] X. Devroey, M. Cordy, P. Schobbens, A. Legay, and P. Heymans. State machine flattening, a mapping study and tools assessment. In **2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**, pages 1–8, April 2015.

[61] Xavier Devroey. VIBeS Case Studies: Featured Transition Systems and Feature Models. https://doi.org/10.5281/zenodo.4105900, October 2020.

[62] Xavier Devroey. VIBeS: Variability Intensive system Behavioral teSting framework. https://github.com/xdevroey/vibes, 2022.

[63] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Hamza Samih, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Statistical prioritization for software product line testing: an experience report. **Software & Systems Modeling**, pages 1–19, 2015.

[64] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Hamza Samih, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Statistical prioritization for

software product line testing: an experience report. **Softw. Syst. Model.**, 16(1):153–171, 2017.

[65] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Pierre-Yves Schobbens, Axel Legay, and Patrick Heymans. Towards Statistical Prioritization for Software Product Lines Testing. In **Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems**, VaMoS '14, New York, NY, USA, 2014. Association for Computing Machinery.

[66] Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Covering SPL Behaviour with Sampled Configurations: An Initial Assessment. In Klaus Schmid, Øystein Haugen, and Johannes Müller, editors, **Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '15, Hildesheim, Germany, January 21-23, 2015**, page 59. ACM, 2015.

[67] Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Search-based similarity-driven behavioural SPL testing. In **Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems**, pages 89–96, 2016.

[68] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Featured Model-based Mutation Analysis. In **Proceedings of the 38th International Conference on Software Engineering**, ICSE '16, pages 655–666, New York, NY, USA, 2016. ACM.

[69] Nicola Di Mauro, Annalisa Appice, and Teresa MA Basile. Activity prediction of business process instances with inception CNN models. In **International conference of the italian association for artificial intelligence**, pages 348–361. Springer, 2019.

[70] Nicolas D'Ippolito, Dario Fischbein, Marsha Chechik, and Sebastian Uchitel. MTSA: The Modal Transition System Analyser. In **2008 23rd IEEE/ACM International Conference on Automated Software Engineering**, pages 475–476. IEEE, 2008.

[71] Johannes Dorn, Sven Apel, and Norbert Siegmund. Generating Attributed Variability Models for Transfer Learning. In **Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems**, VAMOS '20. ACM, 2020.

[72] Edilton Lima dos Santos, Sophie Fortz, Gilles Perrouin, and Pierre-Yves Schobbens. A vision to identify architectural smells in self-adaptive systems using behavioral maps. In Robert Heinrich, Raffaela Mirandola, and Danny Weyns, editors, **4th Context-aware, Autonomous and Smart Architectures International Workshop (CASA 2021)**, 15th European Conference on Software Architecture (ECSA 2021), Växjö, Sweden, September 2021. CEUR Workshop Proceedings.

[73] Edilton Lima dos Santos, Sophie Fortz, Pierre-Yves Schobbens, and Gilles Perrouin. Identifying Architectural Smells in Self-Adaptive Systems at Runtime. In **13ème édition de la Conférence francophone sur les Architectures Logicielles (CAL)**, Vannes, France, June 2022.

[74] Pierre Dupont. Incremental regular inference. In **International Colloquium on Grammatical Inference**, pages 222–237. Springer, 1996.

[75] Holger Eichelberger and Klaus Schmid. Mapping the design-space of textual variability modeling languages: a refined analysis. **International Journal on Software Tools for Technology Transfer**, 17(5):559–584, 2015.

[76] Joerg Evermann, Jana-Rebecca Rehse, and Peter Fettke. Predicting process behaviour using deep learning. **Decision Support Systems**, 100:129–140, 2017.

[77] Dirk Fahland and Wil M.P. van der Aalst. Model repair — aligning process models to reality. **Information Systems**, 47:220–243, 2015.

[78] Alessandro Fantechi and Stefania Gnesi. Formal modeling for product families engineering. In **2008 12th International Software Product Line Conference**, pages 193–202, 2008.

[79] Ederson Carvalhar Fernandes, Barry Fitzgerald, Liam Brown, and Milton Borsato. Machine Learning and Process Mining applied to Process Optimization: Bibliometric and Systemic Analysis. **Procedia Manufacturing**, 38:84–91, 2019. 29th International Conference on Flexible Automation and Intelligent Manufacturing (FAIM 2019), June 24-28, 2019, Limerick, Ireland, Beyond Industry 4.0: Industrial Advances, Engineering Education and Intelligent Manufacturing.

[80] Fabio Ferreira, Luciana Lourdes Silva, and Marco Tulio Valente. Software engineering meets deep learning: a mapping study. In **Proceedings of the 36th Annual ACM Symposium on Applied Computing**, pages 1542–1549, 2021.

[81] Dario Fischbein, Sebastian Uchitel, and Victor Braberman. A foundation for behavioural conformance in software product line architectures. In **Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis**, pages 39–48, 2006.

[82] Sophie Fortz. LIFTS: Learning Featured Transition Systems. In **Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume B, Leicester, United Kindom**, SPLC '21, page 1–6, New York, NY, USA, 2021. Association for Computing Machinery.

[83] Sophie Fortz. Learning featured transition systems, August 2023. Sophie Fortz is supported by the FNRS via a FRIA grant.

[84] Sophie Fortz. Variability-aware Behavioural Learning. In **Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume B, Tokyo, Japan**, SPLC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[85] Sophie Fortz, Fred Mesnard, Etienne Payet, Gilles Perrouin, Wim Vanhoof, and Germán Vidal. An SMT-Based Concolic Testing Tool for Logic Programs. In Keisuke Nakano and Konstantinos Sagonas, editors, **15th International Symposium on Functional and Logic Programming (FLOPS 2020), Akita, Japan**, pages 215–219, Cham, sep 2020. Springer International Publishing.

[86] Sophie Fortz, Paul Temple, Xavier Devroey, Patrick Heymans, and Gilles Perrouin. VaryMinions: leveraging RNNs to identify variants in event logs. In Apostolos Ampatzoglou, Daniel Feitosa, Gemma Catolino, and Valentina Lenarduzzi, editors, **Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution, Athens, Greece, 23 August 2021**, pages 13–18. ACM, 2021.

[87] Sophie Fortz, Paul Temple, Xavier Devroey, Patrick Heymans, and Gilles Perrouin. VaryMinions. https://zenodo.org/record/7492126, December 2022. Sophie Fortz is supported by the FNRS via a FRIA grant. Gilles Perrouin is an FNRS Research Associate.

[88] Vanderson Hafemann Fragal, Adenilso Simao, and Mohammad Reza Mousavi. Validated test models for software product lines: Featured finite state machines. In **Formal Aspects of Component Software: 13th International Conference, FACS 2016, Besançon, France, October 19-21, 2016, Revised Selected Papers**, pages 210–227. Springer, 2017.

[89] Vanderson Hafemann Fragal, Adenilso Simao, and Mohammad Reza Mousavi. Hierarchical featured state machines. **Science of Computer Programming**, 171:67–88, 2019.

[90] Vanderson Hafemann Fragal, Adenilso Simao, Mohammad Reza Mousavi, and Uraz Cengiz Turker. Extending HSI test generation method for software product lines. **The Computer Journal**, 62(1):109–129, 2019.

[91] Gordon Fraser and Andreas Zeller. Exploiting Common Object Usage in Test Case Generation. In **2011 Fourth IEEE International Conference on Software Testing, Verification and Validation**, pages 80–89. IEEE, mar 2011.

[92] Salvador García, Daniel Molina, Manuel Lozano, and Francisco Herrera. A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the CEC'2005 special session on real parameter optimization. **Journal of Heuristics**, 15(6):617–644, 2009.

[93] Stuart Geman, Elie Bienenstock, and René Doursat. Neural networks and the bias/variance dilemma. **Neural computation**, 4(1):1–58, 1992.

[94] Salah Ghamizi, Maxime Cordy, Mike Papadakis, and Yves Le Traon. Automated search for configurations of convolutional neural network architectures. In **Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A**, pages 119–130. ACM, 2019.

[95] Salah Ghamizi, Maxime Cordy, Mike Papadakis, and Yves Le Traon. FeatureNET: diversity-driven generation of deep learning models. In **Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings**, pages 41–44. ACM, 2020.

[96] Javad Ghofrani, Ehsan Kozegar, Arezoo Bozorgmehr, and Mohammad Divband Soorati. Reusability in artificial neural networks: an empirical study. In **Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B**, pages 122–129. ACM, 2019.

[97] Javad Ghofrani, Ehsan Kozegar, Anna Lena Fehlhaber, and Mohammad Divband Soorati. Applying product line engineering concepts to deep neural networks. In **Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A**, pages 72–77. ACM, 2019.

[98] Arthur Gill et al. Introduction to the theory of finite-state machines, 1962.

[99] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In **Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation**, pages 213–223, 2005.

[100] E Mark Gold. Language identification in the limit. **Information and control**, 10(5):447–474, 1967.

[101] Priyanka Golia, Mate Soos, Sourav Chakraborty, and Kuldeep S. Meel. Designing Samplers is Easy: The Boon of Testers. In **Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021**, pages 222–230. IEEE, 2021.

[102] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. **Deep learning**. MIT press, 2016.

[103] Joel Greenyer, Christian Brenner, Maxime Cordy, Patrick Heymans, and Erika Gressi. Incrementally Synthesizing Controllers from Scenario-based Product Line Specifications. In **Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering**, ESEC/FSE 2013, pages 433–443, New York, NY, USA, 2013. ACM.

[104] Joel Greenyer, Amir Molzam Sharifloo, Maxime Cordy, and Patrick Heymans. Efficient consistency checking of scenario-based product-line specifications. In **2012 20th IEEE International Requirements Engineering Conference (RE)**, pages 161–170. IEEE, 2012.

[105] Huong Ha and Hongyu Zhang. Deepperf: performance prediction for configurable software with deep sparse neural network. In **2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)**, pages 1095–1106. IEEE, 2019.

[106] Vanderson Hafemann Fragal, Adenilso Simao, and Mohammad Reza Mousavi. Validated Test Models for Software Product Lines: Featured Finite State Machines. In Olga Kouchnarenko and Ramtin Khosravi, editors, **Formal Aspects of Component Software**, pages 210–227, Cham, 2017. Springer International Publishing.

[107] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. **Empir. Softw. Eng.**, 24(2):674–717, 2019.

[108] Robert J Hall. Feature combination and interaction detection via foreground/background models. **Computer Networks**, 32(4):449–469, 2000.

[109] Xue Han, Lianxue Hu, Lijun Mei, Yabin Dang, Shivali Agarwal, Xin Zhou, and Pengwei Hu. A-BPS: Automatic Business Process Discovery Service using Ordered Neurons LSTM. In **2020 IEEE International Conference on Web Services (ICWS)**, pages 428–432. IEEE, 2020.

[110] Khadijah Muzzammil Hanga, Yevgeniya Kovalchuk, and Mohamed Medhat Gaber. A Graph-Based Approach to Interpreting Recurrent Neural Networks in Process Mining. **IEEE Access**, 8:172923–172938, 2020.

[111] Nitin Harane and Sheetal Rathi. Comprehensive survey on deep learning approaches in predictive business process monitoring. **Modern Approaches in Machine Learning and Cognitive Science: A Walkthrough**, pages 115–128, 2020.

[112] Eva Hariyanti, Arif Djunaidy, and Daniel Siahaan. Information security vulnerability prediction based on business process model using machine learning approach. **Computers & Security**, 110:102422, 2021.

[113] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. **IEEE Trans. Software Eng.**, 40(7):650–670, 2014.

[114] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. PLEDGE: a product line editor and test generation tool. In **17th International Software Product Line Conference co-located workshops, SPLC 2013 workshops, Tokyo, Japan - August 26 - 30, 2013**, pages 126–129. ACM, 2013.

[115] Steffen Herbold, Patrick Harms, and Jens Grabowski. Combining usage-based and model-based testing for service-oriented architectures in the industrial practice. **International Journal on Software Tools for Technology Transfer**, 19(3):309–324, jun 2017.

[116] Marijn JH Heule and Sicco Verwer. Software model synthesis using satisfiability solvers. **Empirical Software Engineering**, 18(4):825–856, 2013.

[117] Markku Hinkka, Teemu Lehto, Keijo Heljanko, and Alexander Jung. Classifying process instances using recurrent neural networks. In **International Conference on Business Process Management**, pages 313–324. Springer, 2018.

[118] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. **International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems**, 6(02):107–116, 1998.

[119] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. **Neural computation**, 9(8):1735–1780, 1997.

[120] Falk Howar, Dimitra Giannakopoulou, and Zvonimir Rakamarić. Hybrid learning: interface generation through static, dynamic, and symbolic analysis. In **Proceedings of the 2013 International Symposium on Software Testing and Analysis**, pages 268–279. ACM, 2013.

[121] Falk Howar, Bengt Jonsson, and Frits Vaandrager. Combining black-box and white-box techniques for learning register automata. **Computing and Software Science. LNCS**, 10000, 2018.

[122] Like Hui and Mikhail Belkin. Evaluation of neural architectures trained with square loss vs cross-entropy in classification tasks. In **The Ninth International Conference on Learning Representations (ICLR 2021)**, 2021.

[123] Ahmet Iscen, Giorgos Tolias, Yannis Avrithis, and Ondrej Chum. Label propagation for deep semi-supervised learning. In **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**, pages 5070–5079, 2019.

[124] Paul Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. **Bulletin de la Société Vaudoise des Sciences Naturelles**, 37:547–579, 1901.

[125] Nathalie Japkowicz and Mohak Shah. **Evaluating learning algorithms: a classification perspective**. Cambridge University Press, 2011.

[126] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. Properties of realistic feature models make combinatorial testing of product lines feasible. In **Model Driven Engineering Languages and Systems: 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings 14**, pages 638–652. Springer, 2011.

[127] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. The interplay of sampling and machine learning for software performance prediction. **IEEE Software**, 37(4):58–66, 2020.

[128] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A S Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University, Software Engineering Institute, 1990.

[129] Kyo Chul Kang. Jubilee Celebration Panel: Past, Present, Future of SPL and SPLC. In **SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kingdom, September 6-11, 2021**. ACM, September 2021.

[130] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. FeatureIDE: A tool framework for feature-oriented software development. In **2009 IEEE 31st International Conference on Software Engineering**, pages 611–614. IEEE, 2009.

[131] Robert M Keller. Formal verification of parallel programs. **Communications of the ACM**, 19(7):371–384, 1976.

[132] Stuart Kent. Model Driven Engineering. In **IFM**, pages 286–298, London, UK, 2002. Springer-Verlag.

[133] Mohamed Lamine Kerdoudi, Tewfik Ziadi, Chouki Tibermacine, and Salah Sadou. Recovering Software Architecture Product Lines. In **2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)**, pages 226–235. IEEE, 2019.

[134] Seyedehzahra Khoshmanesh and Robyn Lutz. Does Link Prediction Help Find Feature Interactions in Software Product Lines? In **2020 IEEE Seventh International Workshop on Artificial Intelligence for Requirements Engineering (AIRE)**, pages 87–90. IEEE, 2020.

[135] Kamran Kowsari, Donald E Brown, Mojtaba Heidarysafa, Kiana Jafari Meimandi, Matthew S Gerber, and Laura E Barnes. HDLTex: Hierarchical deep learning for text classification. In **16th IEEE international conference on machine learning and applications (ICMLA)**, pages 364–371. IEEE, 2017.

[136] Wolfgang Kratsch, Jonas Manderscheid, Maximilian Röglinger, and Johannes Seyfried. Machine learning in business process monitoring: a comparison of deep learning and classical approaches used for outcome prediction. **Business & Information Systems Engineering**, pages 1–16, 2020.

[137] Marcello La Rosa and Marlon Dumas. Configurable process models: how to adopt standard practices in your how way? **BPTrends Newsletter**, 2008.

[138] K Lang. Evidence driven state merging with search. **Rapport technique TR98–139, NECI**, 31, 1998.

[139] Kevin Lang, Barak A Pearlmutter, and Rodney Price. Results of the Abbadingo one DFA learning competition and a new evidence driven state merging algorithm. In **Fourth International Colloquium on Grammatical Inference (ICGI-98)**, volume 98, 1998.

[140] Kevin J Lang. Random DFA's can be approximately learned from sparse uniform examples. In **Proceedings of the fifth annual workshop on Computational learning theory**, pages 45–52, 1992.

[141] Kim G Larsen and Bent Thomsen. A modal process logic. In **Proceedings Third Annual Symposium on Logic in Computer Science**, pages 203–204. IEEE Computer Society, 1988.

[142] Dong-Hyun Lee et al. Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks. In **Workshop on challenges in representation learning, ICML**, 2013.

[143] Maikel Leemans, Wil M. P. van der Aalst, and Mark G. J. van den Brand. The Statechart Workbench: Enabling scalable software event log analysis using process mining. In **2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)**, pages 502–506. IEEE, mar 2018.

[144] Keqin Li, Roland Groz, and Muzammil Shahbaz. Integration testing of components guided by incremental state machine learning. In **Testing: Academic & Industrial Conference-Practice And Research Techniques (TAIC PART'06)**, pages 59–70. IEEE, 2006.

[145] Yang Li, Sandro Schulze, and Gunter Saake. Reverse Engineering Variability from Natural Language Documents: A Systematic Literature Review. In **SPLC'17 - Volume A**, SPLC '17, pages 133–142, New York, NY, USA, 2017. ACM.

[146] Yang Li, Sandro Schulze, and Jiahua Xu. Feature Terms Prediction: A Feasible Way to Indicate the Notion of Features in Software Product Line. In **Proceedings of the Evaluation and Assessment in Software Engineering**, EASE '20, page 90–99, New York, NY, USA, 2020. Association for Computing Machinery.

[147] Crescencio Lima, Wesley KG Assunção, Jabier Martinez, William Mendonça, Ivan C Machado, and Christina FG Chavez. Product line architecture recovery with outlier filtering in software families: the Apo-Games case study. **Journal of the Brazilian Computer Society**, 25(1):1–17, 2019.

[148] Edilton Lima dos Santos, Sophie Fortz, Pierre-Yves Schobbens, and Gilles Perrouin. Behavioral Maps: Identifying Architectural Smells in Self-adaptive Systems at Runtime. In Patrizia Scandurra, Matthias Galster, Raffaela Mirandola, and Danny Weyns, editors, **Software Architecture**, pages 159–180, Cham, 2022. Springer International Publishing.

[149] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal Loss for Dense Object Detection. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, 42(2):318–327, 2020.

[150] Martin Lippert and Stephen Roock. **Refactoring in large software projects: performing complex restructurings successfully**. John Wiley & Sons, 2006.

[151] Pengfei Liu, Xipeng Qiu, and Xuanjing Huang. Recurrent Neural Network for Text Classification with Multi-Task Learning. In **Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence**, IJCAI'16, page 2873–2879. AAAI Press, 2016.

[152] Roberto E. Lopez-Herrejon, Lukas Linsbauer, and Alexander Egyed. A systematic mapping study of search-based software engineering for software product lines. **Information and Software Technology**, 61:33 – 51, 2015.

[153] Ronny S Mans, MH Schonenberg, Minseok Song, Wil MP van der Aalst, and Piet JM Bakker. Application of process mining in healthcare–a case study in a Dutch hospital. In **International joint conference on biomedical engineering systems and technologies**, pages 425–438. Springer, 2008.

[154] Hugo Martin, Mathieu Acher, Juliana Alves Pereira, and Jean-Marc Jézéquel. A comparison of performance specialization learning for configurable systems. In **Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume A**, pages 46–57, 2021.

[155] Jabier Martinez and Ali Parsai. D3.1: Identification of relevant state of the art. Technical report, ITEA 3 ReVAMP2 Project Consortium, 2018.

[156] Martin Matzner and Bjoern Eskofier. Time Matters: Time-Aware LSTMs for Predictive Business Process Monitoring. In **Process Mining Workshops: ICPM 2020 International Workshops, Padua, Italy, October 5–8, 2020, Revised Selected Papers**, volume 406, page 112. Springer Nature, 2021.

[157] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. On essential configuration complexity: Measuring interactions in highly-configurable systems. In **Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering**, pages 483–494, 2016.

[158] Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next generation LearnLib. In **International Conference on Tools and Algorithms for the Construction and Analysis of Systems**, pages 220–223. Springer, 2011.

[159] Raphaël Michel, Andreas Classen, Arnaud Hubaux, and Quentin Boucher. A Formal Semantics for Feature Cardinalities in Feature Diagrams. In **Proceedings of the Fifth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'11), Namur, Belgium, January 27-29**, pages 83–90. ACM Press, 2011.

[160] Gabriela K Michelon, Lukas Linsbauer, Wesley KG Assunção, Stefan Fischer, and Alexander Egyed. A Hybrid Feature Location Technique for Re-engineering Single Systems into Software Product Lines. In **15th International Working Conference on Variability Modelling of Software-Intensive Systems**, pages 1–9, 2021.

[161] Edward F Moore et al. Gedanken-experiments on sequential machines. **Automata studies**, 34:129–153, 1956.

[162] Johann Mortara and Philippe Collet. **Capturing the Diversity of Analyses on the Linux Kernel Variability**, page 160–171. Association for Computing Machinery, New York, NY, USA, 2021.

[163] Edi Muškardin, Bernhard K Aichernig, Ingo Pill, Andrea Pferscher, and Martin Tappler. AALpy: an active automata learning library. **Innovations in Systems and Software Engineering**, 18(3):417–426, 2022.

[164] Thiloshon Nagarajah and Guhanathan Poravi. A review on automated machine learning (AutoML) systems. In **2019 IEEE 5th International Conference for Convergence in Technology (I2CT)**, pages 1–6. IEEE, 2019.

[165] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. Using bad learners to find good configurations. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, **Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017**, pages 257–267. ACM, 2017.

[166] Peter Bjorn Nemenyi. **Distribution-free multiple comparisons.** Princeton University, 1963.

[167] Dominic A Neu, Johannes Lahann, and Peter Fettke. A systematic literature review on state-of-the-art deep learning methods for process prediction. **Artificial Intelligence Review**, pages 1–27, 2021.

[168] Hoang Thi Cam Nguyen, Suhwan Lee, Jongchan Kim, Jonghyeon Ko, and Marco Comuzzi. Autoencoders for improving quality of process event logs. **Expert Systems with Applications**, 131:132–147, 2019.

[169] Oliver Niese. **An integrated approach to testing complex systems**. PhD thesis, Technical University of Dortmund, Germany, 2003.

[170] Timo Nolle, Alexander Seeliger, and Max Mühlhäuser. BINet: multivariate business process anomaly detection using deep learning. In **International Conference on Business Process Management**, pages 271–287. Springer, 2018.

[171] Timo Nolle, Alexander Seeliger, Nils Thoma, and Max Mühlhäuser. DeepAlign: Alignment-Based Process Anomaly Correction Using Recurrent Neural Networks. In **International Conference on Advanced Information Systems Engineering**, pages 319–333. Springer, 2020.

[172] José Oncina and Pedro Garcia. Inferring regular languages in polynomial updated time. In **Pattern recognition and image analysis: selected papers from the IVth Spanish Symposium**, pages 49–61. World Scientific, 1992.

[173] Gyunam Park and Minseok Song. Predicting performances in business processes using deep neural networks. **Decision Support Systems**, 129:113191, 2020.

[174] Juliana Alves Pereira, Hugo Martin, Paul Temple, and Mathieu Acher. Machine Learning and Configurable Systems: A Gentle Introduction. In **Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A**, SPLC '20. ACM, 2020.

[175] Gilles Perrouin, Mathieu Acher, Jean-Marc Davril, Axel Legay, and Patrick Heymans. A complexity tale: web configurators. In **Proceedings of the 1st International Workshop on Variability and Complexity in Software Design**, pages 28–31, 2016.

[176] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In **2010 Third international conference on software testing, verification and validation**, pages 459–468. IEEE, 2010.

[177] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. **Software Product Line Engineering: Foundations, Principles and Techniques**. Springer, 2005.

[178] PureSystems. Pure::Variants Website http://www.pure-systems.com/, 2011.

[179] Harald Raffelt and Bernhard Steffen. Learnlib: A library for automata learning and experimentation. In **International Conference on Fundamental Approaches to Software Engineering**, pages 377–380. Springer, 2006.

[180] Belén Ramos-Gutiérrez, Ángel Jesús Varela-Vaca, José A Galindo, María Teresa Gómez-López, and David Benavides. Discovering configuration workflows from existing logs using process mining. **Empirical Software Engineering**, 26(1):1–41, 2021.

[181] Marcello La Rosa, Wil MP Van Der Aalst, Marlon Dumas, and Fredrik P Milani. Business process variability modeling: A survey. **ACM Computing Surveys**, 50(1):1–45, 2017.

[182] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. **nature**, 323(6088):533–536, 1986.

[183] Ana Belén Sánchez, Sergio Segura, José Antonio Parejo, and Antonio Ruiz Cortés. Variability testing in the wild: the Drupal case study. **Softw. Syst. Model.**, 16(1):173–194, 2017.

[184] D.K. Sasidharan and S.K. N. **Full Stack Development with JHipster: Build full stack applications and microservices with Spring Boot and modern JavaScript frameworks, 2nd Edition**. Packt Publishing, 2020.

[185] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. **Comput. Networks**, 51(2):456–479, 2007.

[186] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. **Computer Networks**, 51(2):456–479, 2007.

[187] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. **IEEE transactions on Signal Processing**, 45(11):2673–2681, 1997.

[188] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. **ACM SIGSOFT Software Engineering Notes**, 30(5):263–272, 2005.

[189] Muzammil Shahbaz and Roland Groz. Inferring Mealy machines. In **FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings 2**, pages 207–222. Springer, 2009.

[190] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. The Variability Model of The Linux Kernel. **VaMoS**, 10(10):45–51, 2010.

[191] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. Reverse engineering feature models. In **Proceedings of the 33rd International Conference on Software Engineering**, pages 461–470. ACM, 2011.

[192] Liwei Shen, Xin Peng, Jindu Liu, and Wenyun Zhao. Towards feature-oriented variability reconfiguration in dynamic software product lines. In **Top Productivity through Software Reuse: 12th International Conference on Software Reuse, ICSR 2011, Pohang, South Korea, June 13-17, 2011. Proceedings 12**, pages 52–68. Springer, 2011.

[193] Sharon Shoham, Eran Yahav, Stephen J Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. **IEEE Transactions on Software Engineering**, 34(5):651–666, 2008.

[194] Yangyang Shu, Yulei Sui, Hongyu Zhang, and Guandong Xu. Perf-AL: Performance prediction for configurable software through adversarial learning. In **Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**, pages 1–11, 2020.

[195] Rabab Sikal, Hanae Sbai, and Laila Kjiri. Configurable process mining: variability Discovery Approach. In **IEEE 5th International Congress on Information Science and Technology (CiSt)**, pages 137–142. IEEE, 2018.

[196] Larissa Rocha Soares, Jens Meinicke, Sarah Nadi, Christian Kästner, and Eduardo Santana de Almeida. VarXplorer: Lightweight process for dynamic

analysis of feature interactions. In **Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems**, pages 59–66, 2018.

[197] Minseok Song, H Yang, Seyed Hossein Siadat, and Mykola Pechenizkiy. A comparative study of dimensionality reduction techniques to enhance trace clustering performances. **Expert Systems with Applications**, 40(9):3722 – 3737, 2013.

[198] Sara E Sprenkle, Lori L Pollock, and Lucy M Simko. Configuring effective navigation models and abstract test cases for web applications by analysing user behaviour. **Software Testing, Verification and Reliability**, 23(6):439–464, 2013.

[199] Stefan Strüder, Mukelabai Mukelabai, Daniel Strüber, and Thorsten Berger. Feature-oriented defect prediction. In **Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A**, pages 1–12. ACM, 2020.

[200] Xiaoxiao Sun, Wenjie Hou, Yuke Ying, and Dongjin Yu. Remaining Time Prediction of Business Processes based on Multilayer Machine Learning. In **2020 IEEE International Conference on Web Services (ICWS)**, pages 554–558. IEEE, 2020.

[201] Shaghayegh Tavassoli, Carlos Diego N Damasceno, Ramtin Khosravi, and Mohammad Reza Mousavi. Adaptive behavioral model learning for software product lines. In **Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume A**, pages 142–153, 2022.

[202] Shaghayegh Tavassoli, Carlos Diego N Damasceno, Mohammad Reza Mousavi, and Ramtin Khosravi. A benchmark for active learning of variability-intensive systems. In **Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume A**, pages 245–249, 2022.

[203] Niek Tax, Ilya Verenich, Marcello La Rosa, and Marlon Dumas. Predictive Business Process Monitoring with LSTM Neural Networks. In **Advanced Information Systems Engineering**, pages 477–492. Springer, 2017.

[204] Farbod Taymouri, Marcello La Rosa, Marlon Dumas, and Fabrizio Maria Maggi. Business process variant analysis: Survey and classification. **Knowledge-Based Systems**, 211:106557, 2021.

[205] Edgar Tello-Leal, Jorge Roa, Mariano Rubiolo, and Ulises M Ramirez-Alcocer. Predicting activities in business processes with LSTM recurrent neural networks. In **2018 ITU Kaleidoscope: Machine Learning for a 5G Future (ITU K)**, pages 1–7. IEEE, 2018.

[206] Paul Temple, Mathieu Acher, and Jean-Marc Jézéquel. Empirical Assessment of Multimorphic Testing. **IEEE Transactions on Software Engineering**, 47(7):1511–1527, 2021.

[207] Paul Temple, José Angel Galindo, Mathieu Acher, and Jean-Marc Jézéquel. Using machine learning to infer constraints for product lines. In Hong Mei, editor, **Proceedings of the 20th International Systems and Software Product Line Conference, SPLC 2016, Beijing, China, September 16-23, 2016**, pages 209–218. ACM, 2016.

[208] Paul Temple and Gilles Perrouin. Explicit or Implicit? On Feature Engineering for ML-Based Variability-Intensive Systems. In **Proceedings of the 17th International Working Conference on Variability Modelling of Software-Intensive Systems**, VaMoS '23, page 91–93, New York, NY, USA, 2023. Association for Computing Machinery.

[209] Paul Temple, Gilles Perrouin, Mathieu Acher, Battista Biggio, Jean-Marc Jézéquel, and Fabio Roli. Empirical assessment of generating adversarial configurations for software product lines. **Empirical Software Engineering**, 26(1):1–49, 2021.

[210] Maurice H ter Beek, Ferruccio Damiani, Stefania Gnesi, Franco Mazzanti, and Luca Paolini. From featured transition systems to modal transition systems with variability constraints. In **Software Engineering and Formal Methods: 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings**, pages 344–359. Springer, 2015.

[211] Maurice H. ter Beek, Ferruccio Damiani, Stefania Gnesi, Franco Mazzanti, and Luca Paolini. On the expressiveness of modal transition systems with variability constraints. **Science of Computer Programming**, 169:1–17, 2019.

[212] Maurice H ter Beek, Ferruccio Damiani, Michael Lienhardt, Franco Mazzanti, and Luca Paolini. Efficient static analysis and verification of featured transition systems. **Empirical Software Engineering**, 27(1):10, 2022.

[213] Maurice H ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. Modelling and analysing variability in product families: model checking of modal transition systems with variability constraints. **Journal of Logical and Algebraic Methods in Programming**, 85(2):287–315, 2016.

[214] Paolo Tonella, Alessandro Marchetto, Cu Duy Nguyen, Yue Jia, Kiran Lakhotia, and Mark Harman. Finding the optimal balance between over and under approximation of models inferred from execution logs. In **Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation, ICST 2012**, pages 21–30. IEEE, 2012.

[215] Paolo Tonella, Roberto Tiella, and Cu Duy Nguyen. Interpolated n-grams for model based testing. In **Proceedings of the 36th International Conference on Software Engineering - ICSE 2014**, pages 562–572. ACM Press, 2014.

[216] Frits Vaandrager. Model Learning. **Commun. ACM**, 60(2):86–95, January 2017.

[217] Pavel Valov, Jianmei Guo, and Krzysztof Czarnecki. Empirical comparison of regression methods for variability-aware performance prediction. In **Proceedings of the 19th International Conference on Software Product Line**, pages 186–190, 2015.

[218] Wil van der Aalst. **Process Mining: Data Science in Action**. Springer Publishing Company, Incorporated, 2nd edition, 2016.

[219] Wil MP Van der Aalst. Business process management: a comprehensive survey. **International Scholarly Research Notices**, 2013, 2013.

[220] Wil MP Van Der Aalst, Arthur HM Ter Hofstede, and Mathias Weske. Business process management: A survey. **Business process management**, 2678(1019):1–12, 2003.

[221] Wil MP Van der Aalst, Boudewijn F van Dongen, Christian W Günther, Anne Rozinat, HMW Verbeek, and AJMM Weijters. Prom: The process mining toolkit. In **Proceedings of the BPM 2009 Demonstration Track (BPMDemos 2009, Ulm, Germany, September 8, 2009)**, pages 1–4. CEUR-WS. org, 2009.

[222] B.F. (Boudewijn) van Dongen. BPI Challenge 2015, May 2015.

[223] Boudewijn van Dongen. BPI Challenge 2020, Mar 2020.

[224] Boudewijn F Van Dongen, Ana Karla A de Medeiros, HMW Verbeek, AJMM Weijters, and Wil MP van Der Aalst. The prom framework: A new era in process mining tool support. In **Applications and Theory of Petri Nets 2005: 26th International Conference, ICATPN 2005, Miami, USA, June 20-25, 2005. Proceedings 26**, pages 444–454. Springer, 2005.

[225] Ángel Jesús Varela-Vaca, José A Galindo, Belén Ramos-Gutiérrez, María Teresa Gómez-López, and David Benavides. Process mining to unleash variability management: discovering configuration workflows using logs. In **Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A**, pages 265–276. ACM, 2019.

[226] Mahsa Varshosaz, Lars Luthmann, Paul Mohr, Malte Lochau, and Mohammad Reza Mousavi. Modal transition system encoding of featured transition systems. **Journal of Logical and Algebraic Methods in Programming**, 106:1–28, 2019.

[227] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. White-box analysis over machine learning: Modeling performance of configurable systems. In **2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)**, pages 1072–1084. IEEE, 2021.

[228] Ishwar Venugopal, Jessica Töllich, Michael Fairbank, and Ansgar Scherp. A Comparison of Deep-Learning Methods for Analysing and Predicting Business Processes. **arXiv preprint arXiv:2102.07838**, 2021.

[229] Sicco Verwer and Christian A Hammerschmidt. flexfringe: A Passive Automaton Learning Package. In L O'Conner, editor, **2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)**, pages 638–642. IEEE, sep 2017.

[230] Michele Volpato and Jan Tretmans. Approximate active learning of nondeterministic input output transition systems. **Electronic Communications of the EASST**, 72, 2015.

[231] Geetika Vyas, Sonali Vyas, Prasanta Kumar Paul, Amita Sharma, and Chitra Bhardwaj. Prediction algorithms and consecutive estimation of software product line feature model usability. In **2019 Amity International Conference on Artificial Intelligence (AICAI)**, pages 774–777. IEEE, 2019.

[232] Neil Walkinshaw, Kirill Bogdanov, Christophe Damas, Bernard Lambeau, and Pierre Dupont. A framework for the competitive evaluation of model inference techniques. In **Proceedings of the First International Workshop on Model Inference In Testing**, pages 1–9, 2010.

[233] Jiaojiao Wang, Dongjin Yu, Chengfei Liu, and Xiaoxiao Sun. Outcome-oriented predictive process monitoring with attention-based bidirectional LSTM neural networks. In **2019 IEEE International Conference on Web Services (ICWS)**, pages 360–367. IEEE, 2019.

[234] Max Weber, Sven Apel, and Norbert Siegmund. White-Box Performance-Influence models: A profiling and learning approach. In **2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)**, pages 1059–1071. IEEE, 2021.

[235] Markus Weckesser, Roland Kluge, Martin Pfannemüller, Michael Matthé, Andy Schürr, and Christian Becker. Optimal reconfiguration of dynamic software product lines based on performance-influence models. In **Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1**, pages 98–109, 2018.

[236] M. Welsing, J. Maetschke, K. Thomas, A. Gützlaff, G. Schuh, and S. Meusert. Combining process mining and machine learning for lead time prediction in high variance processes. In Bernd-Arno Behrens, Alexander Brosius, Wolfgang Hintze, Steffen Ihlenfeldt, and Jens Peter Wulfsberg, editors, **Production at the leading edge of technology**, pages 528–537, Berlin, Heidelberg, 2021. Springer Berlin Heidelberg.

[237] Pamela Zave. **An experiment in feature engineering**, pages 353–377. Springer New York, New York, NY, 2003.

[238] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. Performance prediction of configurable software systems by Fourier learning. In **2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)**, pages 365–373. IEEE, 2015.