UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Improving model learning by inferring separating sequences from traces

Rafael dos Santos Braz

Dissertação de Mestrado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP
SÃO CARLOS

Rafael dos Santos Braz

Improving model learning by inferring separating sequences from traces

Dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP – in accordance with the requirements of the Computer and Mathematical Sciences Graduate Program, for the degree of Master in Science. *EXAMINATION BOARD PRESENTATION COPY*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Adenilso da Silva Simão

USP – São Carlos
May 2023

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

```
        Braz, Rafael dos Santos
B827i      Improving model learning by inferring separating
        sequences from traces / Rafael dos Santos Braz;
        orientador Adenilso da Silva Simão. -- São Carlos,
        2023.
           83 p.


           Dissertação (Mestrado - Programa de Pós-Graduação
        em Ciências de Computação e Matemática
        Computacional) -- Instituto de Ciências Matemáticas
        e de Computação, Universidade de São Paulo, 2023.


           1. Inferência de modelos. 2. Aprendizagem
        híbrida. 3. Máquinas de Estados Finitos. I. Simão,
        Adenilso da Silva, orient. II. Título.
```

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Rafael dos Santos Braz

Aprimorando a inferência de modelos por meio da seleção de sequências de separação a partir de exemplos do comportamento de sistemas

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional. *EXEMPLAR DE DEFESA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Adenilso da Silva Simão

USP – São Carlos
Maio de 2023

This dissertation is dedicated to all the researchers who, despite all the adversities, persisted in seeking knowledge.

ACKNOWLEDGEMENTS

I want to thank my family, who were always there to support me in this journey, in particular my mother and father, who kept encouraging me to do research, and my little brother, who was always there beside me during the nights of writing this dissertation. I also want to thank my advisor Adenilso Simão for always being comprehensive, available, reliable, and the best for putting me on the right track.

I also want to give special thanks to Roland Groz and Catherine Oriat for being highly valuable to this research project and for making all of this possible.

Also, I would like to thank Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) and Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) for partially financing this project, and Laboratoire d'Informatique de Grenoble (LIG) for hosting me in France for conducting this research.

RESUMO

BRAZ, R. S. **Aprimorando a inferência de modelos por meio da seleção de sequências de separação a partir de exemplos do comportamento de sistemas**. 2023. 83 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2023.

Modelos capazes de representar o comportamento de sistemas, como uma Máquina de Estado Finitos (MEF), são essenciais para o desenvolvimento e a manutenção de software, pois servem de base para várias atividades automatizadas, tais como teste, verificação, validação e refinamento de sistemas. Em contrapartida a sua importância, modelos geralmente são complexos e custosos para se obter. Uma opção para amenizar esse problema é a inferência de modelos, que permite inferir automaticamente, ou com pouca interação humana, um modelo que represente o comportamento do sistema. Esse processo pode ser classificado principalmente em *inferência passiva* (infere modelos a partir de exemplos do comportamento de um sistema) e *inferência ativa* (infere modelos a partir da interação com o sistema). Nesta dissertação, é proposto um método para inferir sequências de separação a partir de *traces* (exemplos observados previamente do comportamento do sistema) e aplicá-las para aprimorar o processo de inferência de modelos. Uma sequência de separação é uma sequência de símbolos de entrada capaz de distinguir um par de estados distintos de uma MEF ao produzir sequências de saída diferentes para cada estado. Quando um conjunto de sequências de separação distingue todos os pares de estados distintos em uma MEF, ele é chamado de conjunto de caracterização, ou W-set. O método proposto recebe um conjunto de *traces* e os processa para extrair todas as suas subsequências de comprimento k, criando uma estrutura de dados chamada W-tree que resume as observações relevantes do comportamento do sistema indicado nos *traces*. O resultado do método é um conjunto das n melhores sequências de separação que um algoritmo de inferência de modelo pode aplicar para aprimorar seu W-set e seu processo de inferência. O método proposto foi implementado, integrado a um algoritmo de *inferencia ativa* chamado hW-inference, e um estudo de caso foi conduzido, no qual foram empregados 40 *traces* diferentes. Como principal resultado do experimento, foi observado que o método proposto pode melhorar o processo de aprendizagem em 24%, em média, e em até 48% em seu melhor caso.

Palavras-chave: Máquinas de Estados Finitos, inferência de modelos, aprendizagem híbrida.

ABSTRACT

BRAZ, R. S. **Improving model learning by inferring separating sequences from traces**. 2023. 83 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2023.

Models that can represent the behavior of systems, such as a Finite State Machine (FSM), are crucial for software development and maintenance as they serve as a base for several automated activities like testing, verification, validation, and refinement of systems. Contrasting their importance and value, models are usually complex and costly to obtain. One option to mitigate this problem is model inference which provides the possibility to automatically, or at least with little human interaction, learn a model that represents the behavior of a system. This process can be mainly classified into *passive inference* (builds models from examples of the behavior of a system) and *active inference* (builds models from interacting with the system). In this dissertation, we propose a method for learning separating sequences from traces (examples of a previously observed behavior of the system) and applying it to improve the process of model inference. A separating sequence is an input sequence capable of distinguishing a pair of distinct states of a machine by yielding different output sequences for each state. When a set of separating sequences distinguishes all pairs of distinct states in the FSM, it is called a characterization set, or W-set. Our proposed method receives a set of traces, processes them to extract all their k-length subsequences, and uses them to build a data structure called W-tree that summarizes the relevant observations of the system's behavior indicated in the traces. The method's output is a set of the n-best separating sequences that a model inference algorithm applies to improve its W-set and its inference process. We implemented our proposed method, integrated it with an *active inference* algorithm called hW-inference, and performed a case study in which we used 40 different traces. We observed that the proposed method could improve the learning process by 24%, on average, and up to 48% in the best-case setting.

Keywords: Finite State Machines, model inference, hybrid learning.

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AFR	Residual Finite Automata
DFSM	Deterministic Finite State Machine
DFSMs	Deterministic Finite State Machines
EFSM	Extended Finite State Machine
FSM	Finite State Machine
FSMs	Finite State Machines
MAT	Minimally Adequate Teacher
MBT	Model-Based Testing
NDFSM	Non-Deterministic Finite State Machine
NDFSMs	Non-Deterministic Finite State Machines
RPNI	Regular Positive Negative Inference
SRS	Software Requirements Specification
SUT	*System Under Test*
V&V	Verification and Validation

CONTENTS

CHAPTER

1

INTRODUCTION

1.1 Contextualization

Models are crucial for the development and maintenance of software systems. In Software Engineering, models are commonly used to describe software requirements and behavior. Models capable of representing software behavior serve as a starting point for automated methods for several activities, e.g., testing, verification, validation, and refinement of systems (WALKINSHAW *et al.*, 2013; Naeem Irfan; ORIAT; GROZ, 2013; NOVELLA; TUFO; FIENGO, 2018).

Although such models are important for Software Engineering and ease the automation of numerous testing and software quality assurance activities, they are usually unavailable in practice. The main obstacle to the existence of models in practice is the non-triviality of their construction and maintenance. It is worth mentioning that, especially in the context of agile methods, the requirements and the implementation are constantly changing, making the process of building and applying models by hand difficult (MEINKE; WALKINSHAW, 2012).

One option to mitigate this problem is model inference, which enables one to infer a model representing a system's behavior automatically, or at least with little human interaction. To accomplish this task, it is valid to discover the behavioral model directly from the source code or examples of the system's behavior (WALKINSHAW *et al.*, 2008; Naeem Irfan; ORIAT; GROZ, 2013). Inference by examples of behavior is based on the concept of black-box. Approaches in this category do not have access to the program's source code and need to infer a model only through observations of the system's behavior obtained either previously or through interactions with the system (BOLLIG *et al.*, 2009).

However, there is a conflict between the benefits of applying models for software specification and testing and the inherent high cost of their creation and maintenance in practice. To mitigate this problem, automated model inference can be used. A well-known model of behavior is the so-called Finite State Machine, which has been used in Software Engineering for software specification for decades (DAVIS, 1988; FUJIWARA *et al.*, 1991; PACHAROEN *et al.*, 2013).

Model inference techniques that perform learning directly on source code rely on access to the code itself, a fact that provides both advantages and disadvantages since being able to analyze the code of a program provides valuable information about its internal structure, but it is increasingly common for programs to be distributed as executable black boxes and therefore without access to their internal structure (Naeem Irfan; ORIAT; GROZ, 2013).

Within the context of model inference, the *active learning* approach depends on a communication channel with the system so that interacting and testing it to learn its behavior is possible, which generates costs for its preparation and execution, besides there are situations in which such communication is not possible (BOLLIG *et al.*, 2009; GROZ *et al.*, 2020). Another possibility is passive model inference, which does not require access to the system since it is executed based on a finite set of examples of the system's behavior, which can be positive or negative examples. Positive examples demonstrate proper behavior exhibited by the system, and negative examples are understood as counterexamples, which do not correspond to the system's behavior (HEERDT, 2016; COOK; WOLF, 1998). It is worth noting that counterexamples are considered optional since they are not always available in practice. In scenarios where examples are collected automatically from the system, it is usual that there are no negative examples available (COOK; WOLF, 1998).

1.2 Objectives

This dissertation fits into the context of model inference, and its main objectives are centered on it. As mentioned above, software models are crucial for several activities in software development and testing; however, obtaining and maintaining them implies a high cost. Model inference mitigates this problem, but it is still an expensive task.

In Finite State Machine inference, separating sequences and characterization sets is important. A separating sequence is an input sequence capable of distinguishing a pair of states of a machine by yielding different output sequences for each state. That is the reason separating sequences are also called distinguishing sequences. When a set of separating sequences can distinguish all pairs of states of a machine, it is called a characterization set or W-set. Model inference algorithms can apply W-sets to localize the states of the machine they are learning and check for inconsistencies. Characterization sets can affect the effectiveness and the cost of the model inference algorithms, especially *active learning* approaches (HALM, 2021).

Our primary goal in this dissertation is to reduce model learning's overall cost. We aim to propose a method for learning separating sequences from traces (examples of a previously observed behavior of the system, which are constituted of input/output sequences) so that we can improve the characterization sets used by the inference algorithms to make the process more efficient.

As specific goals of this dissertation, we can list the following: 1) describe and implement the proposed method for separating sequences learning, 2) integrate the proposed method into a *active learning* Finite State Machine inference algorithm called hW-inference, 3) make use of the *passive learning* data available (traces) to improve the *active learning* algorithm, 4) prepare and conduct a case study to assess the impacts of the proposed method on the process of model inference.

To assess the impact of the proposed method on the model inference process, we performed a case study called *Scanette*, which represents a deterministic FSM model of a supermarket self-service scanning system with 121 states, 15 inputs, and more than 1800 transitions. In the experiment, we used 40 different traces, and 800 executions of the model inference method were performed. As key findings, we can indicate that the proposed method can indeed improve the learning process since we observed an average improvement of 24% on average and up to 48% in the best-case setting.

This dissertation is organized as follows. Chapter 2 contextualize software models in Software Engineering, detailing model inference and introducing model-based testing. Chapter 3 deeper in the context of model learning, but specifically for Finite State Machines. Chapter 4 detail the main contribution of this dissertation by presenting the proposed method, called W-inference. Chapter 5 presents the experimental evaluation of the proposed method and discuss its results. Last, Chapter 6 concludes this dissertation and suggests future work.

CHAPTER 2

SOFTWARE MODELS

2.1 Initial Considerations

Software models, also named computation models, have been applied in Software Engineering for decades for software specification and testing. Models, especially those capable of representing the behavior of systems, have become fundamental concepts for developing and maintaining software systems as they serve as a basis for automating numerous approaches, such as testing, verification, validation, and refinement of systems (WALKINSHAW *et al.*, 2013).

Software models and their inference are the core of this dissertation. It is essential to comprehend the concepts of behavioral models and the key aspects of passive and active inference of models. This chapter introduces this topic, which is used later in the dissertation.

This chapter is organized as follows. Section 2.2 discuss and contextualize the purpose of software models in Software Engineering. Section 2.2.2 introduces model inference and its characteristics. Lastly, Section 2.2.3 summarizes the model-based testing concepts.

2.2 Software Models

The Systems and Software Engineering Vocabulary (ISO/IEC/IEEE..., 2017) establish a set of definitions for describing a model; we can summarize them in three complementary definitions: 1) "*related collection of instances of meta-objects, representing (describing or prescribing) an information system, or parts thereof, such as a software product*"; 2) "*semantically closed abstraction of a system or a complete description of a system from a particular perspective*"; 3) "*representation of a system of interest, from the perspective of a related set of concerns*".

Computation models are crucial for developing and maintaining computational systems, as highlighted by Walkinshaw *et al.* (2013). In the context of Software Engineering, models are commonly used to describe software requirements or software behavior. Models capable of representing software behavior benefit Software Engineering as they can be the starting point for automating software testing, verification, validation, and refinement methods.

According to Davis (1988), software models that represent the behavior of systems, especially the ones that represent their external behavior, are valuable instruments for system development, even in early stages; therefore, they are helpful not only in situations where the system is already in operation. It is common to create a Software Requirements Specification (SRS) document during the software requirements specification stage, describing the expected behavior of the system of interest in detail. Such description is fundamental for software testing; however, it is usually expressed in natural language, which can lead to ambiguity, inconsistency, and incompleteness. In this scenario, formal software models become a convenient option that can be processed computationally and soften the problems of natural languages.

Davis (1988) indicates "rules" that formal specification methods should follow. Those rules are summarized as follows.

- The technique should provide a basis for automated prototype and system test generation;
- The technique should provide automated checks for ambiguity, incompleteness, and inconsistency;
- The technique should describe the system's behavior in terms of external product behavior, not internal product components;
- The technique should be suitable to the particular application.

The main computation models, by the above criteria, are described in the following section.

2.2.1 Models of Software External Behavior

As mentioned above, techniques capable of representing the external behavior of a system serve as the basis for several software development activities. Due to its importance and diversity, this section highlights some of the approaches available in the literature. The following sections briefly describe them.

2.2.1.1 Finite State Machines

Since Finite State Machines (FSMs) are the primary model of this dissertation, Chapter 3 is dedicated to them. Therefore, here we only provide a brief description and example for comparison with the other techniques presented in this section.

FSMs are models used to describe behavior through states and transitions. A Finite State Machine (FSM) receives external stimuli (inputs) and produces outputs. The states of an FSM represent all the possible situations the machine can or will be in a given moment (WAGNER *et al.*, 2006). Transitions based on the current state and the provided input are triggered to move from one state to another and produce outputs (DAVIS, 1988).

For instance, Figure 1 shows an FSM represented graphically by a graph in which each vertex is a state, and each edge is an input/output transition. In the example machine, there are three distinct states (I_0, I_1, I_2) accepting a set of inputs $X = \{a, b, c\}$ and producing a set of outputs $Y = \{e, f\}$. Assuming that the current state is I_0, it is necessary to receive an input $x \in X$ to determine the next state since the FSM can accept b or a as an input and transfer to state I_1 – producing an output f or e, respectively – or accept c and transfer to I_2, producing e as the output. We can interpret the other states following the same logic.

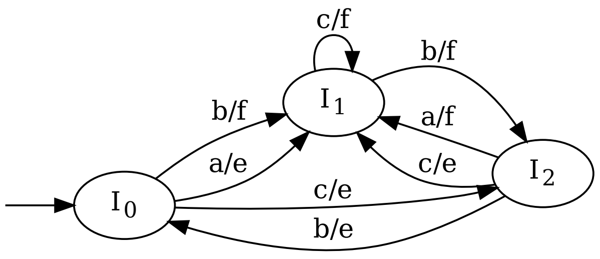

Figure 1 – An example of FSM represented by a graph.

Source: Adapted from Fujiwara *et al.* (1991).

2.2.2 Model Inference

According to Naeem Irfan, Oriat and Groz (2013) and Novella, Tufo and Fiengo (2018), several software verification and validation techniques depend on models of external software behavior. Although such models are important in Software Engineering and base the automation of many testing and quality assurance activities, it is often difficult to find them in practice. The main obstacle to using models in practice is their non-trivial creation and maintenance. Besides, there are scenarios where copyrights in the model do not allow sharing and usage by other organizations or automated tools. Complementary to the non-triviality of manually creating and maintaining models, it is unrealistic to expect its developers to maintain models of systems, especially in an agile software development context where both requirements and implementation are continuously changing (MEINKE; WALKINSHAW, 2012).

One option to mitigate this problem is model inference which provides the possibility to automatically, or at least with little human interaction, learn a model that represents the behavior of a system. Discovering the behavioral model directly from source code is valid, as in the work of Walkinshaw *et al.* (2008), which employs symbolic execution of the code to build an adequate model of its behavior. Techniques that are executed directly on the source code depend on the access to the code itself, a fact that provides advantages and disadvantages since being able to analyze the code of a program provides valuable information about its internal structure; however, it is increasingly common that programs are distributed as executable black boxes and therefore without access to their internal structure (Naeem Irfan; ORIAT; GROZ, 2013).

Given the increasing unavailability of source code, it is possible to infer behavioral models only through interactions with system components. In scenarios where interaction with the system is impossible, another source of information is required, such as existing specifications, contact with specialists, or program execution records.

In the context of model inference, we can distinguish two main approaches:

- **Passive inference**: this approach, also known as passive, by samples or offline learning, builds models from examples of the behavior of a system. Algorithms in this category take as input a finite and fixed set of positive and negative examples. Assuming that the language to be inferred is $\boldsymbol{L}$, the positive examples belong to $\boldsymbol{L}$, and, on the other hand, the negative examples do not belong to $\boldsymbol{L}$ and usually are optional (BOLLIG *et al.*, 2009; COOK; WOLF, 1998; HEERDT, 2016).

- **Active inference**: this approach, also known as active or online learning, aims to learn behavioral models from interaction with a system. In this case, the learning algorithms can communicate with the system, interact, and test it to learn its behavior. Furthermore, active inference can start with a set of examples, but new examples and counterexamples must be acquired from interacting with the system to classify learning as active or, more precisely, hybrid (BOLLIG *et al.*, 2009; PACHAROEN *et al.*, 2013; GROZ *et al.*, 2020).

As suggested by Naeem Irfan, Oriat and Groz (2013), these two approaches have different inputs, advantages, and disadvantages since passive learning has the advantage of not needing any communication with the system or other sources of information beyond the set of examples provided; in the other hand, it has the disadvantage of its accuracy being limited by the quality of the set of examples provided that are unlikely to be able to expose the scenario as a whole, which makes the inferred model only an approximation of reality. In the case of active learning, this context is inverted so that its accuracy is potentially higher, but a means of interaction with the system is essential.

A concept from model and language inference is the property of identification in the limit, proposed by Gold (1967). In short, if an inference algorithm presents the identification in the limit property, it will provide a correct and adequate model if it has access to sufficient information. This property was formally proposed in the context of language inference, but we can extend it to other formal model inference scenarios in Software Engineering.

We complement and deepen the concepts discussed in this section in the context of FSM inference in Section 3.3. The following section discusses the characteristics that an inferred model should present to be considered good.

2.2.2.1 Characteristics of a good-inferred model

Usually, building an infinite set of models representing the same scenario for the same set of events is possible. Indeed, not all models in this set align with the goals and expectations of the inference. Consequently, it is indispensable that there are parameters that help to define what a good model is. Cook and Wolf (1998) suggest three main characteristics that an inferred model, in the context of passive inference, should present:

- *Fully accounts the behavior examples provided.* The inferred model must reject the negative examples if they are available and fully accept the positive examples. Furthermore, the behavior examples considered must be subjected to noise restrictions in the data when the inference method includes noise treatment.
- *Successfully identifies patterns made up of sequencing, selection, and iteration.* The inferred model must be able to complement the provided examples by identifying and applying patterns; therefore, a good model should recognize more than just the set of provided examples.
- *Does not needlessly complicate the patterns identified.* The inferred model should be as simple as possible, within the limits of feasibility, without unnecessary additional elements being added to the final model. An ideal model should not complicate the identified patterns with extra states and transitions in models based on them.

In the theoretical field of FSMs, linking two different terms to a good passively-inferred Finite State Machine is common: accuracy and minimality. Accuracy conforms to the first characteristic mentioned above, and minimality, in turn, conforms to the last characteristic. However, the minimality generally implies that the inferred FSM contains only the smallest necessary number of states, which makes it a stricter rule than the one listed by Cook and Wolf (1998).

It is worth emphasizing that minimality and accuracy are conflicting terms since maximum accuracy needs a more significant number of states and redundancy so that all the examples provided have unique paths from the machine's initial state, and maximum minimality requires only one state that recognizes all possible inputs. Therefore, the last feature listed above aims to soften this conflicting relationship to infer a feasible model.

Complementary, Petrenko *et al.* (1994) indicate the conformance relations (see Section 3.2.2.1 for formal definition) that are useful when one needs to compare two FSMs. In the context of FSM inference, the pair of FSMs consists of the specification machine (the real FSM representing the system's behavior under analysis) and the inferred FSM. The two most essential conformance relations in FSM inference are reduction and equivalence. Reduction implies that the language (all input/output sequences accepted by the FSM) recognized by the inferred FSM must be contained in the language recognized by the specification FSM, i.e., all input/output sequences accepted by the inferred FSM must also be accepted by the specification FSM. The equivalence relation is stronger than the reduction relation, as equivalence establishes that the two FSMs must recognize the same language.

Therefore, a good-inferred FSM should follow the principles indicated by Cook and Wolf (1998), discussed above, and be a reduction of its specification.

2.2.3 Model-Based Testing

As discussed in Section 2.2, software models are closely related to software specification and testing activities. This section presents an introduction to software testing and model-based testing concepts.

Software testing usually belongs to a broader topic known as Verification and Validation (V&V), in which the term verification refers to tasks that ensure that the software correctly implements a given functionality, and the term validation enclosures tasks that ensure that the software was created according to the client's requirements (PRESSMAN, 2011). From another perspective, verification and validation aim to answer the questions "Are we building the product correctly?" and "Are we building the correct product?", respectively. Even with different views, both perspectives still relate V&V and software testing to software quality assurance.

As suggested by Utting and Legeard (2007), software testing can be interpreted as an activity aimed at evaluating the quality of a software product and improving it by identifying defects and problems. Logically, just pointing out the problems is not enough to increase the quality of a product, but it is a fundamental step for their correction.

Naeem Irfan, Oriat and Groz (2013) indicates that the software testing activity can be performed manually or automatically, although automated tests are the main focus of research in the area. An interesting and more complete definition of software testing encompasses four concepts in a single sentence: "*Software testing consists of the **dynamic** verification that the program provides **expected** behaviors on a **finite** set of test cases, suitably **selected** from the usually infinite execution domain*" (BOURQUE; FAIRLEY, 2014, p. 4-1). The four mentioned concepts (in bold) provide us with interesting implications. The testing activity is always conducted on a running program, although some communities admit to using software testing for static activities. The term *expected* refers to the fact that it is essential to verify and decide whether the observed outputs are adequate and expected. In addition, the term *finite* refers to the large number – often infinite – of test cases (set composed of input data, execution conditions, and expected results (ISO/IEC/IEEE..., 2017)) that can be elaborated for the same system, making exhaustive testing (verification of the entire set) impractical. Therefore, the testing activity should focus on only a subset of the possible test cases, which must be carefully prioritized and selected.

Various approaches and trends are available in the literature to develop and conduct software testing. According to Pressman (2011), the different testing strategies have the same generic characteristics, although providing different testing models. Software testing starts at the component level and progresses toward the system as a whole. Different software testing techniques are suitable for different Software Engineering approaches and at different points in time. Testing is done by the software developer and an independent group when the project is large enough.

Figure 2 contains an interesting visual representation that correlates several essential concepts in the software testing universe, making it a valuable tool for discussion. The representation follows a three-dimensional Cartesian plane. The transparent straight parallelogram indicates the dimensions of Model-Based Testing (MBT), the branch of software testing that motivates this work. The 'SUT Scale' axis displays the scale of the *System Under Test* (SUT), from small units to the system as a whole. The 'Characteristics being tested' axis indicates different aspects that can be tested. The 'Source of testing ...' axis indicates the information available to elaborate and conduct the test.

MBT is a well-established branch of software testing in the literature. The origins of the field can be found in the work of Moore (1956), entitled "Gedanken-Experiments on Sequential Machines,", which used FSMs. However, a technique that applied model testing and inference was not proposed until the work of Weyuker (1983).

Utting, Pretschner and Legeard (2012) define MBT as an activity that includes the processes and techniques for automatically deriving abstract test cases from abstract systems models, generating concrete tests from abstract tests, and executing (manually

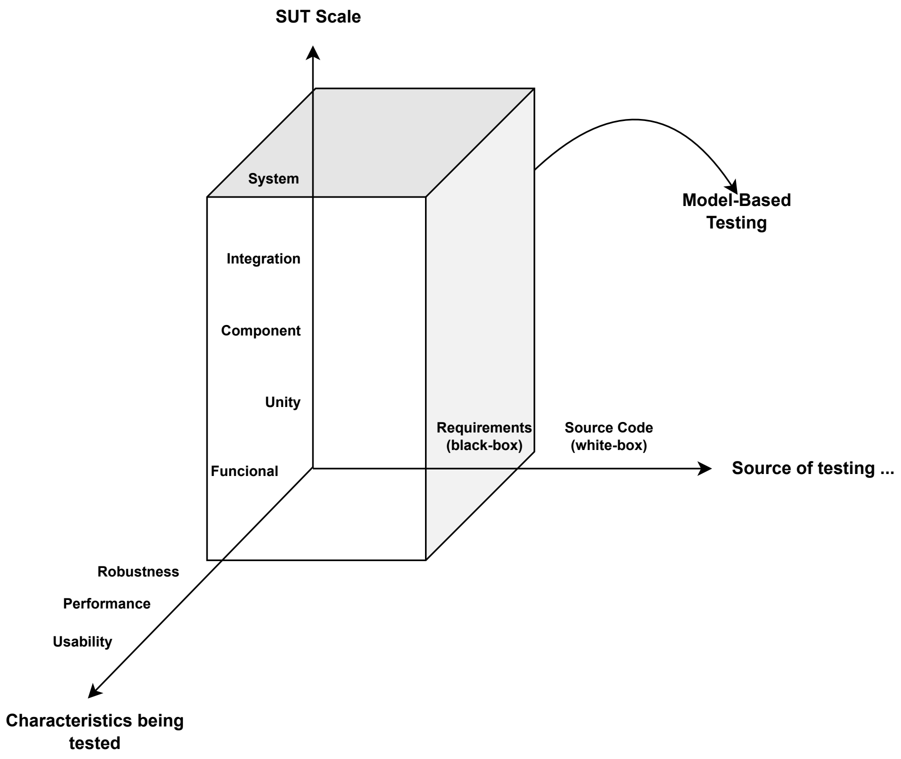

Figure 2 – Dimensions of MBT.

Source: Adapted from Pinheiro (2014).

or automatically) concrete tests. Thus, the software models must be written in sufficiently precise languages so that, in general, a machine can derive test cases from them.

To better understand the dimensions of MBT, we can refer again to Figure 2. The parallelogram represents MBT within the universe of software testing. It is worth noting which aspects are covered by MBT. MBT is comprehensive enough to be applied to all scales of the SUT, from testing small units to the entire system. MBT is traditionally used in the functional testing of systems. Still, its use in other testing categories is also possible, respecting the possible limitations of the type of testing and the adopted technique. Finally, MBT is known to be based on black-box testing, as it is based on the requirements and functionalities of the system and not on its internal structure.

Pressman (2011) emphasizes that MBT requires five steps to be executed, described below. These steps are also represented in Figure 3.

1. **Analysis of an existing behavior model or creation of one**. If an appropriate behavior model already exists, studying it may be sufficient. In the absence of one, creating it manually or automatically through some model inference technique is possible.

2. **Traversing the behavior model and specifying the inputs that will force the software to transition from one state to another**. It is necessary to identify

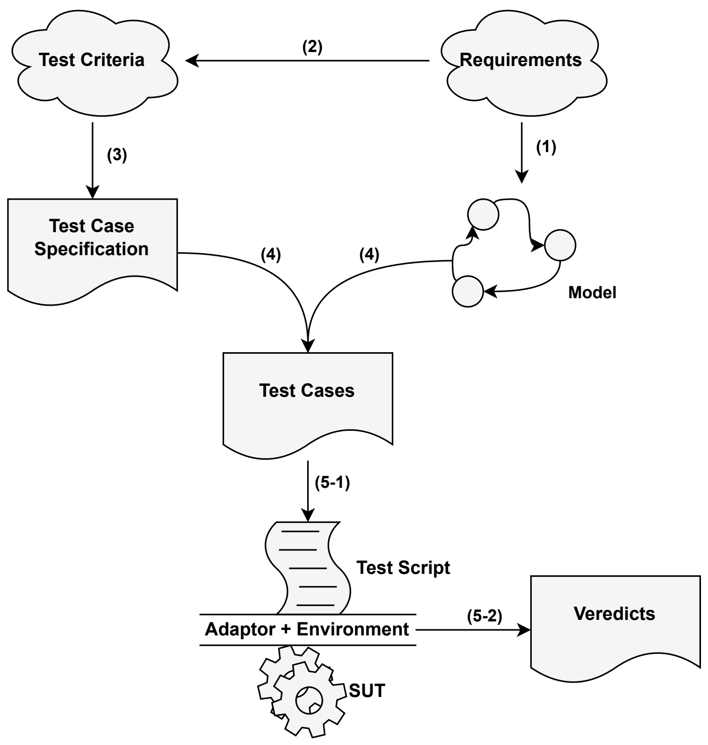

Figure 3 – Visual representation of MBT process.

Source: Adapted from Utting, Pretschner and Legeard (2012).

which inputs trigger the events that make the transitions of the model occur.

3. **Reviewing the behavior model and observing the expected outputs as the software transitions from one state to another**. After specifying the inputs in the previous step, it is necessary to complete the test cases by determining the expected outputs by applying the inputs.

4. **Execution of test cases**. Once the test cases have been established, they can be executed on the SUT. This activity can be conducted manually or through scripts in an automated testing tool.

5. **Comparison of actual and expected results and taking necessary corrective action**. After running the tests, verifying whether the expected results were obtained is possible. If the results differ, the situation should be analyzed, and if necessary, the system, model, or testing apparatus should be corrected.

2.3 Final Considerations

This chapter introduces the concept of software models and contextualizes it within the scope of Software Engineering. The models of the system's external behavior are discussed. In addition, we introduce the activity of inferring models in this chapter and finalize it in Section 3.3. Finally, the discussion is complemented by the topics of software testing and model-based testing, given that the use of models in Software Engineering is usually closely related to testing activities.

CHAPTER

3

FINITE STATE MACHINE INFERENCE

3.1 Initial Considerations

Finite State Machines are a model type that aims to represent behavior through states and transitions. This model type has been used for decades to specify and test computer systems. One aspect that has gained increasing attention is the inference of FSMs, which can discover a Finite State Machine from interacting with a system or a finite set of previous observations of its execution.

The key contribution of this dissertation is in the context of FSM inference; thus, this chapter details the inference of software models and their characteristics, introduced in the previous chapter, but specifically in the context of FSMs.

This chapter is organized as follows. Section 3.2 presents the fundamentals and characteristics of deterministic and non-deterministic FSMs. Finally, Section 3.3 discusses the process of active and passive inference of FSMs.

3.2 Finite State Machines

Finite State Machines are computation models that describe behavior through a hypothetical state-based machine that receives input stimuli and produces outputs. An FSM has as a characteristic the presence of states that represent the possible situations in which the machine can or will be at a certain moment. Another fundamental characteristic of this model type is its state set's finiteness, which restricts the machine to a scenario where the number of distinguishable situations in which the machine can be at a given moment is finite (WAGNER *et al.*, 2006). For the machine to produce an output, functions are purely based on the current state and the received input (DAVIS, 1988).

According to Simão, Petrenko and Maldonado (2009), this type of computation model based on states and transitions has been widely used in several areas, such as hardware design, language recognition, protocol compliance testing, and object-oriented software testing. Historically, Davis (1988) indicates that FSMs were already effectively used to specify requirements for telephony applications, real-time systems, and process control applications. Recently, FSMs – especially non-deterministic ones – have been used to diagnose errors involving human interactions, for example, in rail transport control and other technological solutions that human operators handle (ZHIRABOK; KALININA; SHUMSKII, 2020).

FSMs can be classified as deterministic or non-deterministic, each with its own characteristics and properties. The following sections are dedicated to discussing them.

3.2.1 Deterministic Finite State Machines

Deterministic Finite State Machines (DFSMs) are the FSM type commonly found in the literature. The addendum "deterministic" indicates that, from any state of the Deterministic Finite State Machine (DFSM) and a given input, it is only possible to find a single transition, which redirects the machine to a single state and has a single associated output.

As indicated by Fujiwara *et al.* (1991), a DFSM M_d can be formally defined as a tuple $(\Sigma, \Delta, S, s_0, \delta, \lambda)$, where:

$\Sigma \neq \emptyset$ is the finite set of input symbols,

$\Delta \neq \emptyset$ is the finite set of output symbols,

$S \neq \emptyset$ is the finite set of states,

$s_0 \in S$ is the initial state,

$\delta : S \times \Sigma \rightarrow S \cup \{\phi\}$ is the transition function, which can include the unspecified state ϕ,

$\lambda : S \times \Sigma \rightarrow \Delta \cup \{\varepsilon\}$ is the output function, which can include the unspecified output ε.

According to Gill (1962), it is possible to represent a DFSM as a directed graph or a state-transition table.

When a labeled directed graph represents a DFSM, a graph $G = (V, A)$ is established, where $V = S$ is a finite set of vertices and A is a finite set of edges. Each $a \in A$ is a tuple $a = (v_i, v_j, r)$, where $v_i, v_j \in V$ and r is a label such as "α/β ", where $\alpha \in \Sigma$ and $\beta \in \Delta$. Also, $v_j = \delta(v_i, \alpha)$ and $\beta = \lambda(v_i, \alpha)$.

The directed graph that represents a DFSM can be described by a visual abstraction such as the one found in Figure 4, where: $\Sigma = \{a, b\}$, $\Delta = \{0, 1\}$, $S = \{1, 2, 3\}$ and $s_0 = 3$. The initial state is symbolized by an arrow with only the target state.

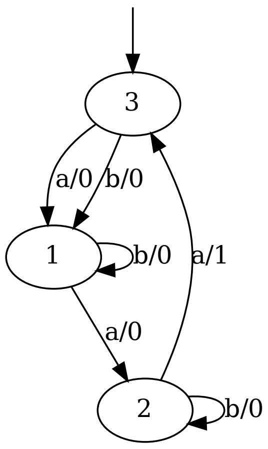

Figure 4 – A DFSM represented as a directed graph.

Source: Adapted from Groz *et al.* (2020).

When a state-transition table represents a DFSM, the functions δ, λ are calculated through the table's rows and columns; in this scenario, the S states demarcate the table rows, and the Σ elements make up the table column headers. To calculate the result of the transition and output functions, it is only necessary to relate the state and input of the desired row and column, respectively, and observe the value of the corresponding cell. Since the FSM in question is deterministic, each table cell must contain a single value, being it a state $s \in S$ for the result of δ or a $x \in \Delta$ for the result of λ, including the values ϕ, ε for unspecified states and outputs, respectively. For example, the DFSM in Figure 4 is represented by a state-transition table in Table 1.

Table 1 – Example of DFSM as a state-transition table.

States	δ		λ	
	Inputs		**Inputs**	
	a	b	a	b
1	2	1	0	0
2	3	2	1	0
3	1	1	0	0

Source: Elaborated by the author.

3.2.1.1 *Properties of Deterministic Finite State Machines*

Fujiwara *et al.* (1991) and Simão, Petrenko and Maldonado (2009) list some relevant Deterministic Finite State Machines properties. These properties are summarized here as follows.

- A DFSM is fully specified if there are transitions for every input from each of its states;

- A DFSM is strongly connected if, for every pair $(s,s') \in S \times S$ of distinct states, there is a sequence of state transitions that leads from s to s';
- Two states s,t of a DFSM M are V-equivalent, denoted by $s \approx_V t$, if, for a certain set V of input sequences, the same output sequences are produced;
- Two states s,t of a DFSM M are equivalent, denoted by $s \approx t$, if, for any set of input sequences V, s,t are V-equivalent;
- Two DFSMs M,M' are equivalent if their initial states are equivalent. Such equivalence can also be found if, for each state of M, there is an equivalent state in M' (DAMASCENO; MASIERO; SIMAO, 2016);
- A DFSM M is minimal if its number of states is less than or equal to any DFSM M' equivalent to M;
- A DFSM is reduced if all pairs $S \times S$ of distinct states are distinguishable;
- A DFSM is initially connected if there is a sequence of state transitions that starts from s_0 and reaches all states of S.

3.2.2 Non-Deterministic Finite State Machines

Although the deterministic version is the most commonly found in the literature, Non-Deterministic Finite State Machines (NDFSMs) are not uncommon computational models for specifying software systems. A Non-Deterministic Finite State Machine (NDFSM) differs from a DFSM by allowing that the same input provides than one acceptable output or reachable target state for a given state (HIERONS, 2004).

While the determinism of a DFSM is helpful in numerous scenarios, since its execution and testing are relatively simplified compared to its non-deterministic version, an NDFSM is more suitable and preferable in specific contexts, such as the specification of systems composed of a series of interacting components. Several subclasses fall into this category of system and have increasingly attracted interest from industry and academia. Communication systems, component-based systems, and service-oriented systems, among others, are examples of this class (PACHAROEN *et al.*, 2013). Another category of intrinsically non-deterministic is the complex reactive systems, which have been modeled as non-deterministic FSMs to mitigate the influence of various external sources on the system (KHALILI; TACCHELLA, 2014).

In this type of system, non-determinism can result from the asynchronous communication process between the system components or from unpredictable activities during the interleaving of components. The intrinsic presence of non-determinism is more naturally modeled and preserved by an NDFSM rather than a DFSM. Moreover, Spitsyna, El-Fakih and Yevtushenko (2007) emphasize that non-determinism can arise from changes in specifications due to performance, flexibility, limited controllability, abstraction, among other factors.

According to Hierons (2003), an NDFSM M_{nd} can be formally defined as a tuple such that $M_{nd} = (S, s_1, h, \Sigma, \Gamma)$, where:

$\Sigma \neq \emptyset$ is a finite set of input symbols,

$\Gamma \neq \emptyset$ is a finite set of output symbols,

$S \neq \emptyset$ is a finite set of states,

$s_1 \in S$ is the initial state,

$h : Q \times \Sigma \rightarrow 2^{Q \times \Gamma}$ is the transition function.

Unlike the deterministic version formalized in the previous section, the NDFSM does not establish an output function since it is assumed that an output $\gamma \in \Gamma$ is produced when executing the function h for a given state and input symbol. The function h is precisely responsible for the "non-deterministic" addendum of this FSM. For each pair $(s \in S, a \in \Sigma)$ given to the function h, a set of pairs $(s' \in S, o \in \Gamma)$ is returned since there is a set of possible transitions that can be executed, and therefore, h does not "determine" the transition to be performed by the machine.

An NDFSM's non-determinism can manifest in three distinct forms, called non-determinism patterns (KHALILI; TACCHELLA, 2014), illustrated in Figure 5. It is valid to distinguish two classes of non-determinism: state non-determinism and output non-determinism. State non-determinism establishes that the same input may lead to different states, while output non-determinism indicates that the same input may produce different outputs. Pattern *(i)* only constitutes state non-determinism. Pattern *(ii)* presents both classes of non-determinism. Finally, pattern *(iii)* indicates only output non-determinism.

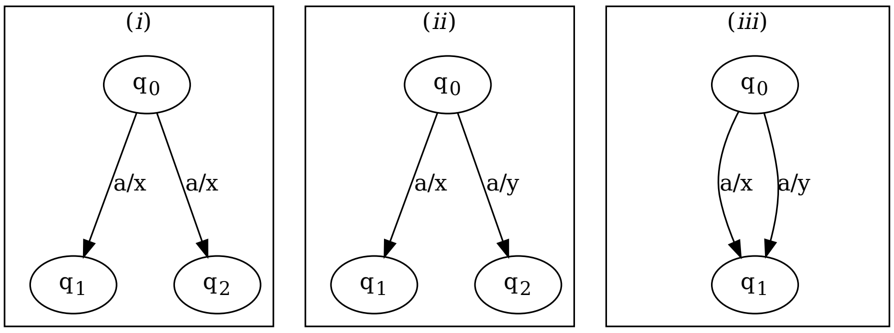

Figure 5 – Example of non-determinism patterns in NDFSMs.

Source: Adapted from Khalili and Tacchella (2014).

Following NDFSMs, it is valid to represent NDFSMs using graphs or state-transition tables.

When an NDFSM is represented using a labeled directed graph, a graph $G = (V, A)$ is established, where $V = S$ is a finite set of vertices and A is a finite set of edges. Each $a \in A$ is a tuple $a = (v_i, v_j, r)$, where $v_i, v_j \in V$ and r is a label such as "α/β", where $\alpha \in \Sigma$ and $\beta \in \Gamma$. The values α and β in the labels are correlated and directly related to the values of v_i and v_j. Thus, the following definitions can be established: $v_j = x | \exists (x \in S, y \in \Gamma) \in h(v_i, \alpha)$ and $\beta = y | \exists (x \in S, y \in \Gamma) \in h(v_i, \alpha)$.

As an illustration of representing NDFSMs as directed graphs, see the NDFSM M_0 in Figure 6, where: $\Sigma = a, b$, $\Gamma = 0, 1$, $S = s_1, s_2, s_3$ and $s_1 = s_1$. An arrow with no origin state symbolizes the initial state, only a destination state. From the figure, it can be observed that non-determinism occurs in the transitions that have $v_i = s_1$ since receiving an input a in the state s_1 allows two possible transitions: $(s_1, s_2, a/0), (s_1, s_3, a/1)$.

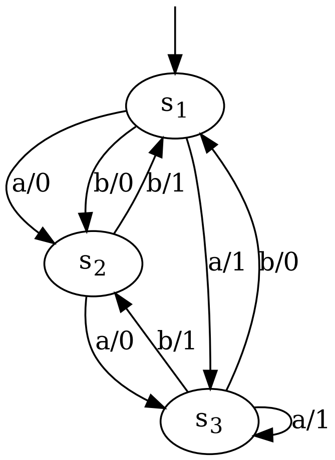

Figure 6 – Example of NDFSM as a graph.

Source: Adapted from Hierons (2003).

By directly adapting the transition table for DFSMs defined in the book by Aho (2007), it is possible to represent NDFSMs using this mechanism. Still, the content of the table cells must be adapted so that the result of the function h is correctly preserved. Therefore, each row of the table is linked to a state $s \in S$, each column is linked to an element α of the input alphabet, and each cell of the table contains a set of ordered pairs (s, y) such that $x \in S, y \in \Gamma$ obtained from the result of $h(s, \alpha)$ for the corresponding row and column. Table 2 contains the representation of the NDFSM from Figure 6 using a state-transition table.

3.2.2.1 Conformance Relations for Finite State Machines

Petrenko *et al.* (1994) define some important and interesting properties, called conformance relations, that are known for being applied in the FSM inference field. Although

Table 2 – Example of an NDFSM as a state-transition table.

States	**Inputs**	
	a	b
s_1	$\{(s_2,0),(s_3,1)\}$	$\{(s_2,0)\}$
s_2	$\{(s_3,0)\}$	$\{(s_1,1)\}$
s_3	$\{(s_3,1)\}$	$\{(s_2,1)\}$

Source: Elaborated by the author.

we summarize the conformance relations here as NDFSM properties, they can also be used for DFSM.

- A NDFSM $M = (S,s_1,h,\Sigma,\Gamma)$ is defined as a specification and $M' = (S',s'1,h',\Sigma,\Gamma)$ is defined as an implementation of the specification M. Therefore, M' is compliant with M if all output sequences produced by M' from all defined input sequences must appear in M;

- M' is called a reduction of M, denoted by the notation $M' \leq M$, when $X^*_M \subseteq X^*_{M'} \wedge \forall \alpha \in X^*_M : h'^*_2(s'_1,\alpha) \subseteq h^*_2(s_1,\alpha)$, where X^β represents the set of all input sequences defined for machine β;

- M' is called distinguishable from M, denoted by the notation $M' \neq_i M$, when $\exists \alpha \in X^*M \wedge \exists y \in h'_2(s'1,\alpha) \wedge y \notin h2(s_1,\alpha)$, that is, M' is not a reduction of M;

- M' is quasi-equivalent to M, denoted by the notation $M' \equiv_D M$, if and only if all output sequences described by M from all defined input sequences must be produced by M', formally $\forall \alpha \in X^M : h'2(s'1,\alpha) = h2^*(s_1,\alpha)$.

The reduction relation can also be defined through language recognition in finite automata: M' is a reduction of M if $L(M') \subseteq L(M)$. For an automaton A, $L(A)$ represents all input sequences accepted by A. Knowing that an NDFSM is a finite automaton for which each transition is also associated with an output alphabet element, it is possible to infer that $L(\beta)$ represents all input/output sequences accepted by NDFSM β (HIERONS, 2003).

In addition to reduction and quasi-equivalence, a stronger relationship can be established for NDFSMs: the so-called equivalence relation, as suggested in (IPATE, 2006). An NDFSM M is equivalent to an NDFSM M' if and only if $M \leq M'$ and $M' \leq M$. Therefore, $L(M') \subseteq L(M)$ and $L(M) \subseteq L(M')$, that is, $L(M') = L(M)$.

3.2.3 Characterization Sets and Separating Sequences

In the context of FSMs, we can highlight two intrinsically related definitions of high importance for this dissertation: characterization sets and their separating sequences.

According to Simão, Petrenko and Maldonado (2009), two states s,t of a DFSM M are distinguishable from each other if there is a separating sequence γ for s,t, such that $\gamma \in \Omega_M(s) \cap \Omega_M(t) \wedge \lambda(s,\gamma) \neq \lambda(t,\gamma)$. The notation $\Omega_F(x)$ represents the set of all input sequences defined in F for the state x, and the function λ is extended here to receive a sequence of inputs instead of a single symbol. The notation $s \not\sim t$ can also describe this property. In other words, a separating sequence is an input sequence capable of distinguishing a pair of states of a machine by yielding different output sequences for each state. That is the reason separating sequences are also called distinguishing sequences.

When a set of separating sequences can distinguish all pairs of states of a DFSM M, it is called a characterization set, or conventionally called a W-set or even a characterizing set. A W-set $W \subset \Sigma^*$ is formally defined as $\forall s,s' \in S : s \neq s' \to \exists w \in W : \lambda(s,w) \neq \lambda(s',w)$ (HALM, 2021).

When there is a minimal DFSM M, there is a characterization set by definition since all pairs of states of M are distinguished by separating sequences.

For example, see Figure 7 where a DFSM with three states (s1, s2, and s3) is presented. We can notice that $W = \{b.b\}$ is enough to distinguish all states of the machine. The blue arrows in the figure represent the output responses of the state for $w \in W$. Each state provides a different output for the same sequence in the W-set: $\lambda(s1,w) = 1.1$, $\lambda(s2,w) = 0.0$, and $\lambda(s3,w) = 0.1$.

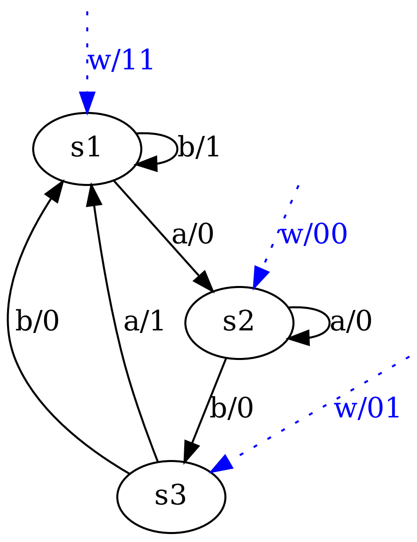

Figure 7 – A DFSM and its W-set representation (in blue).

Source: Adapted from Halm (2021).

3.3 Finite State Machine Inference

As suggested by Pacharoen *et al.* (2013), the use and search for automata learning techniques within formal software verification have grown rapidly in the last two decades. Generally, the goal of these techniques is the inference FSMs. The main advantage of this approach is the possibility of automated inference of FSMs, whose behavior is unknown. The behavior of an FSM can be unknown either because its specification cannot be accessed (or it is non-existent) or the system under analysis can only be seen as a black box whose behavior needs to be discovered and modeled only by interacting with this system, or from an existing set of observations of its execution (GROZ *et al.*, 2020).

In the field of Software Engineering, the use of techniques for automata learning has a significant impact and contributes to the mitigation of various problems, for example, the study of a system with unknown behavior, software testing, properties verification, construction of specifications, maintenance of applications, among others (SHAHBAZ; GROZ, 2009).

Techniques for FSM learning can be classified and grouped according to their characteristics. As these techniques deal with inference, the most fundamental distinction among them is based on active and passive learning approaches, also called online and offline learning (BOLLIG *et al.*, 2009). In the following subsections, these two approaches are discussed.

3.3.1 Active Learning of Finite State Machines

Active learning of automata is not a recent concept since its origins can be found in the work of Gold (1972). Techniques that fit into this category have been available for decades, such as the well-known L^* algorithm, which has received great focus in the literature and was proposed by Angluin (1987).

In this inference approach, learning occurs through queries sent by the learning algorithm (the learner) to a teacher who must be able to answer them correctly (PACHAROEN *et al.*, 2013). The teacher, also known as the Minimally Adequate Teacher (MAT) or oracle, must perfectly know the structure of the model that is desired to be inferred.

As suggested by Groz *et al.* (2020), active learning can be initiated from pre-existing system behavior observations. Still, new evidence is collected from testing the system, and therefore new positive and negative examples will be obtained as needed.

Unlike passive learning, active learning needs to interact with the system. Positively, accessing the source code or the executable file of the system is unnecessary, given that a remote communication channel that allows sending stimuli and obtaining responses is enough to perform the inference.

In the context of deterministic finite automaton inference, it is common for techniques to be based on the L^* algorithm of Angluin (1987), such as the modifications ${L_M}^*$ and ${L_M}^+$ for inferred deterministic FSMs proposed by Shahbaz and Groz (2009), and the modification of Rivest and Schapire (1993). In addition, other techniques are available, such as the recent hW-inference (GROZ *et al.*, 2020) for reactive systems that do not require the operation of reset and the method proposed by Groz *et al.* (2015), also for scenarios without reset. Since hW-inference is the learning algorithm that guides this dissertation, Section 4.2 presents an overview of the method.

The L_N algorithm was proposed by El-Fakih *et al.* (2010) for the inference of fully specified observable NFA and is based on the L^* technique of Angluin (1987). Like in L^*, L_N establishes two query types: the membership query and the equivalence query. A membership query checks whether a given input symbol sequence belongs to the language L recognized by the machine that is desired to be inferred. An equivalence query checks whether a given language L' inferred by the learning algorithm is equivalent to L. If $L' \neq L$, a counterexample is returned by the MAT, indicating a sequence of input symbols that differentiates the two languages. Furthermore, this technique uses the observation table proposed for L^*. An observation table is a data structure the learner uses to store observations (correlations between input and output sequences) and formulate hypotheses about the machine to be inferred (KHALILI; TACCHELLA, 2014).

Based on the L^* technique proposed by Angluin (1987), the L^*_{NM} algorithm was proposed by Pacharoen *et al.* (2013) to refine the L_N algorithm through repetition of k *membership queries* (explicitly extended and called *output queries* to return the output sequence for a given input sequence). The repetition of queries aims to remove the requirement imposed by L_N that the MAT must be able to return all possible outputs in a single query.

The NL^* algorithm was proposed by Bollig *et al.* (2009). It extended the classic L^* algorithm to infer non-deterministic finite automata, more precisely, a subclass called Residual Finite Automata (AFR). Although the machine learned by the algorithm is non-deterministic, it assumes that its underlying system is deterministic.

The N^* technique was proposed by Khalili and Tacchella (2014) and was considered by its creators as "similar in spirit" to the L_N algorithm proposed by El-Fakih *et al.* (2010), with more detailed explanation and empirical evidence.

A new solution was proposed by Petrenko and Avellaneda (2019) for inference of NDFSMs through an approach that differs from the L^* algorithm by not requiring a MAT, equivalence queries, or a conformance tester.

3.3.2 Passive Learning of Finite State Machines

Unlike active learning, passive learning (or offline learning) does not require communication with the system to infer its behavior model. Passive learning of automata depends on a finite set of positive examples and, according to Cook and Wolf (1998), maybe negative ones, which indicates that negative examples are not mandatory but complementary since in scenarios such as observing the execution of a process and its events, negative examples usually do not exist. Positive examples are strings that belong to the language accepted by the automaton of the system's behavior. Negative examples are composed of strings that are not accepted by this language (BOLLIG *et al.*, 2009). Because it only knows the language to be inferred through examples, passive learning is usually known as sample-based learning (HEERDT, 2016). Naturally, when passive learning is applied to infer finite-state machines, the concept of input strings is extended to cover its corresponding output sequences.

Passive learning of automata is not a recent area of study. Classic works such as Gold (1978) have been discussing their difficulties and characteristics for decades. One of the major adversities faced in the passive inference of automata is that it belongs to the NP-Complete class of problems and is described by Gold (1978) as a "computationally difficult" task. However, exciting solutions to the problem exist and continue to be developed, mainly based on heuristics, assumptions about the automaton to be inferred, or seeking approximate solutions. Even after 25 years of the remarks by Gold (1978), Lucas and Reynolds (2003) indicate that inferring automata from examples is still a "difficult task", but interesting and necessary.

Several techniques are available for passive automata inference, such as the solution of Ginsburg (1959), motivated to infer DFSMs for sequential circuit synthesis. In addition, Lucas and Reynolds (2003) discuss the fundamentals of two other approaches: the evolutionary algorithm, based on genetic algorithms and the construction of automata by sequenced mutations, and the evidence-oriented state-merging algorithm, which behaves similarly to automaton minimization techniques. Later, Tsarev and Egorov (2011) present a new solution based on the genetic algorithm of Lucas and Reynolds (2003) seeking optimization. In a relatively different scenario, Petrenko *et al.* (2017) propose an approach capable of acting actively or passively to infer a DFSM through a single example (a single string updated as necessary so that only one machine is produced). It is worth mentioning an efficient algorithm called Regular Positive Negative Inference (RPNI), which has been used to infer deterministic finite automata that are not necessarily minimal, proposed in (LANG, 1992; ONCINA; GARCÍA, 1992). We can also mention the techniques presented in the context of the StaMinA (State Machine Inference Approaches) competition (WALKINSHAW *et al.*, 2013). The results of this competition provided a comparison of efficient techniques for inferring DFSM.

The solution proposed by Biermann and Feldman (1972) can produce non-deterministic and deterministic machines through the same algorithm. This solution operates by characterizing the states by observing future behavior. Given an execution history, the future behavior is determined by the last k events of the sequence. This characteristic makes this method suitable for inferring FSMs from a large set of examples, in which each example has a short sequence of events. This technique is known in the literature and has served as a basis for the creation of other techniques, such as Ktail by Cook and Wolf (1998) and GK-tail proposed by Lorenzoli, Mariani and Pezzè (2008) for inference of a particular type of FSM called Extended Finite State Machine (EFSM), which is commonly non-deterministic and allows transitions to be annotated with constraints and conditions so that both control behavior and data behavior, through variables, can be modeled. Through GK-tail, this solution influenced the method proposed by Walkinshaw, Taylor and Derrick (2016) that also aims to infer an EFSM but only contemplates its deterministic version.

The solution proposed by Das and Mozer (1993), called DOLCE, aims to infer FSMs from a purely statistical perspective, employing neural networks. However, this method is limited in dealing with at most two types of events (two combinations of input/output, for example). To expand the coverage of this technique and facilitate the extraction of the machine produced by the neural network, Cook and Wolf (1998) proposed a method based on DOLCE, called RNet, which will be discussed below. The RNet operates by characterizing states through the observation of their previous behavior. One of the advantages of this approach is the possibility of inferring deterministic and non-deterministic machines through appropriate neural network training.

The method Ktail proposed by Cook and Wolf (1998) is based on the method proposed by Biermann and Feldman (1972). The Ktail method contributes by not requiring a smaller set of examples, producing an FSM with fewer states, and providing an additional parameter that reduces noise interference in the examples.

The Markov method was proposed by Cook and Wolf (1998) and was named as such because it employs Markov models in both a statistical and algorithmic approach. In this scenario, a statistical method is used to find the most probable sequences of events. An algorithmic approach is then used to convert these probabilities into states and transitions of an FSM.

The GK-tail technique proposed by Lorenzoli, Mariani and Pezzè (2008) aims to learn EFSMs and perform inference through a series of mergers of the provided examples, then merges equivalent states in the machine.

The solution proposed by Pradel and Gross (2009) favors the presence of non-determinism, although non-determinism is not explicitly addressed in the paper. The inferred FSM has states and transitions based on probabilistic events.

3.4 Final Considerations

In this chapter, we discussed the main concepts of the FSMs. The foundations, characteristics, formal descriptions, and applications of FSMs were introduced, including the characterization and differentiation of deterministic and non-deterministic FSMs. This chapter also discussed the main concepts and techniques for inferring FSMs, including active and passive learning activities.

CHAPTER

4

W-INFERENCE METHOD

4.1 Initial Considerations

In the previous chapters, we discussed and contextualized model inference, specially FSM inference. In model learning, passive inference learns models from previously collected system behavior examples. This dissertation considers only positive examples representing valid machine behavior excerpts. Positive examples can be characterized as interleaved input/output sequences (traces) in the context of FSMs. We can define a trace ω as $(\Sigma \times \Delta)^* \in L(M)$, where $L(M)$ is the set of input/output sequences accepted by an FSM M, i.e., its language. In contrast, active inference has the advantage of having a communication channel with the system so that the inference algorithm can collect traces by interacting with and testing the system (GROZ *et al.*, 2020). Combining both approaches, hybrid inference merges passive and active inferences so that the learning algorithm can access previously collected traces and interact with the system to collect more examples.

In this chapter, we detail the main contribution of this dissertation: a method for learning separating sequences from traces and applying it to improve the process of model inference.

This chapter is organized as follows. Section 4.2 contextualizes and describes the learning algorithm called hW-inference that the proposed method was originally designed to improve. Section 4.3 details the proposed method. Finally, Section 4.4 discusses the benefits of the relation between the proposed method and hW-inference.

4.2 Overview of hW-inference

Although using the proposed method for different inference algorithms in passive, active, or hybrid inference contexts is possible, we designed and implemented the approach within the hW-inference algorithm, a recent active learning approach.

hW-inference is an inference algorithm that aims to infer an FSM model of the behavior of a black-box system under learning. hW-inference was proposed in (GROZ *et al.*, 2018) and has been extended and improved by different projects such as (GROZ; BREMOND; SIMAO, 2019; GROZ *et al.*, 2020) and (HALM *et al.*, 2022). hW-inference was designed as an active learning approach to produce minimal deterministic FSM models equivalent to the SUL, requiring only that the SUL behaves as a deterministic strongly-connected FSM whose input alphabet is known. Besides, the SUL does not need to provide an input sequence to reset the machine to its initial state as hW-inference can learn the system without resetting it.

The approach proposed in this dissertation is an initial step in turning hW-inference into a hybrid algorithm as we consider previously collected traces as a source of information for better characterizing the states of an FSM during inference.

hW-inference can use two different forms of W-sets and homing sequences: preset and adaptive. The preset version, described in (GROZ *et al.*, 2020) and used in (HALM *et al.*, 2022), applies W-sets and homing sequences as prefixed input sequences, which conforms to the definition by Lee and Yannakakis (1996). In an adaptive context, each input of the sequence to be applied depends on the previous output, so trees are used instead of sets of sequences (LEE; YANNAKAKIS, 1996).

This dissertation considers the preset version of hW-inference since W-sets are sets of sequences that naturally fit the proposed method. In addition, we aim to improve the newest version of hW-inference presented in (HALM *et al.*, 2022), which considers only preset sequences.

Algorithm 1 presents the main algorithm of hW-inference. In the inner **repeat** loop in lines 7-21, we apply the homing sequence h to transfer the machine to a known state. As the algorithm is constantly learning h, it is possible that h is not homing, raising an h-ND inconsistency (h leads to more than one tail state) and indicating the need to refine it. If no inconsistency exists and the tail state of h, indicated in the algorithm as $H(r)$, is not fully characterized yet, a sequence $w \in W$ is applied to better characterize the state. Otherwise, we are ready to learn the transitions of the machine by applying an input symbol from Σ and a sequence from W to characterize the state reached after the transition.

We keep this process until we reach a unique fully-characterized state from the current h. After that, we can learn unknown transitions from the tail state of h. If all transitions of the tail state are already known, we need to find the shortest sequence α from this state that leads to an unknown transition and learn that transition.

The algorithm may find exceptions during the learning, such as h-ND (discussed

Algorithm 1 – Adapted *hw*-inference algorithm

```
1: procedure INFER($\omega$: a given trace)
2:     initialize: $h \leftarrow \varepsilon$; $W \leftarrow \emptyset$
3:     infer $W'$ from $\omega$
4:     $W \leftarrow W \cup W'$
5:     repeat
6:         $S, \lambda, \delta \leftarrow \emptyset$
7:         repeat
8:             apply $h$ and observe $r \in \Delta^*$
9:             if $H(r)$ is undefined then
10:                $H(r) \leftarrow \emptyset$
11:            end if
12:            if $H(r)$ is undefined for some $w \in W$ then
13:                apply $w$, observe $y$:
14:                $H(r) \leftarrow H(r) \cup \{w \mapsto y\}$
15:            else
16:                $s = H(r)$                              ▷ the state reached at end of $h$.
17:                find shortest $\alpha \in \Sigma^*$ leading from $s$ to a state $s' \in S$ with incompletely
    known transition $(s', x)$;
18:                apply $\alpha.x.w$ observe $\beta.o.y$;
19:                $\lambda(s', x) \leftarrow o$ and $\delta(s', x)(w) \leftarrow y$
20:            end if
21:        until $M = (S, \Sigma, \Delta, \delta, \lambda)$ contains a strongly connected complete component
22:        infer $W'$ from $\omega$ and execution trace
23:        $W \leftarrow W \cup W'$
24:        apply $W$-set reduction
25:        ask for a counterexample.
26:        process counterexample as a $W$-ND inconsistency
27:    until no counterexample can be found
28: end procedure
```

above), W-ND (different responses to a sequence in W are found for the same state, indicating that the W-set needs to be refined), counterexamples (an oracle is asked to check whether the current conjecture is equivalent to the machine of the SUL and, if it is not equivalent, a counterexample is provided, exposing a difference in behavior, indicating that we need to start a new sub-inference), among others. The algorithm ends when no inconsistencies or counterexamples can be found during the inference, so the current conjecture is accepted as equivalent to the original machine. The algorithm's exceptions and further explanation can be found in (GROZ *et al.*, 2020).

The blue line (line 24) of the algorithm corresponds to the reassessment of the current W-set (W) aiming to reduce it to a set W' whose level of characterization is the same as W. The W-set reduction was introduced in hW-inference by Halm *et al.* (2022) as a W-set pruning method proposed alongside two other methods. All three methods are described in detail in (HALM *et al.*, 2022). Here we briefly summarize the pruning method to contextualize its benefits for the proposed approach.

When W-set pruning is not applied, W keeps growing during the execution of hW-inference, which leads to an increase of the execution trace (all input/output sequences applied during hW-inference execution) since the number of queries will also increase when states characterizing is necessary. However, keeping all sequences in W is not always needed since parts of sequences, or even entire sequences, can become obsolete during the learning process. To mitigate this problem, we can apply the pruning approach, which consists of two stages.

1. **Removing redundant sequences form W.** If a sequence $w \in W$ can be removed from the W-set so that it is still characterizing, w will be removed,

2. **Shortening sequences**. If a sequence and its last input $w.x \in W$ can be replaced by only its prefix w so that the W-set is still characterizing, x will be removed.

The pruning stages are applied until no sequences fit these conditions.

As an illustration, see the deterministic FSM with three states in Figure 8. Suppose there is a W-set $W = \{a, b, ab, aa\}$. If we apply the first pruning stage, we could remove 'a' and 'ab' since $W' = \{b, aa\}$ is still characterizing. After applying the second pruning stage, we could shorten the sequence 'aa' to 'a', since $W'' = \{b, a\}$ is still characterizing. Table 3 represents the pruning states in this example, in which the elements in red were removed in the first stage, and elements in blue were removed in the second stage.

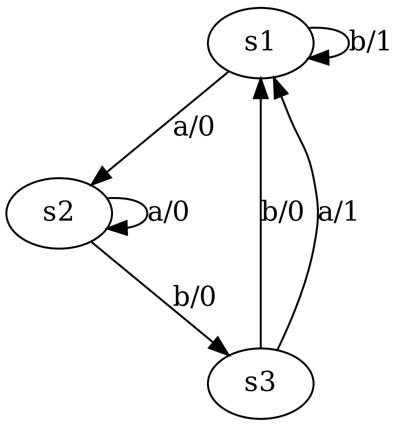

Figure 8 – An simple DFSM with three states.

Source: Adapted from Halm (2021).

Table 3 – Example of W-set reduction.

$w \in W$	States		
	s1	**s2**	**s3**
a	0	0	1
b	1	0	0
ab	00	00	11
aa	00	00	10

Source: Elaborated by the author.

We can use the same FSM in Figure 8 as a simple example of hW-inference algorithm. When we start hW-inference, both h and W are empty, so $h = \varepsilon$ and $W = \emptyset$. If the current state of FSM is s_2, hW-inference would proceed the following way:

$$s_2 \underbrace{\xrightarrow{\varepsilon}}_{h} s_2 \underbrace{\xrightarrow{a/0}}_{x=a} s_2 \underbrace{\xrightarrow{\varepsilon}}_{w=\varepsilon} s_2 \underbrace{\xrightarrow{\varepsilon}}_{h} s_2 \underbrace{\xrightarrow{b/0}}_{x=b} s_3$$

We apply the empty homing sequence $h = \varepsilon$ and the empty W-set $W = \emptyset$, so we learn the state $p_1 = H(\emptyset)$ and the first transition 'a/0' which is a loop in p_1 as W is empty. Similarly, we learn a loop for 'b/0'. As it always happens in the first subinference of hW-inference with empty h and W, we end up with the "daisy" machine in Figure 9.

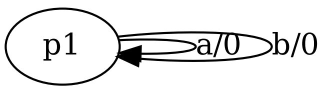

Figure 9 – Conjecture after the first subinference of hW-inference.

Source: Adapted from Halm (2021).

At the end of the first subinference, we ask the oracle for a counterexample, which could be 'a'.

$$s_3 \xrightarrow{a/1} s_1$$

As we apply the counterexample in the current conjecture, an inconsistency in the homing sequence is found since we observed '$h/\varepsilon.a/0$' before and '$h/\varepsilon.a/1$' now. Thus, we must extend the current h with the new input 'a'. Similarly, an W-inconsistency is identified, because both ‘a/0’ and ‘a/1’ were observed from state p_1. In that case, we can extend the current W-set so that $W = \{a\}$.

Once the inconsistencies are solved and $h = a$ is homing, we can proceed to the next subinference.

$$s_1 \underbrace{\xrightarrow{a/0}}_{h} s_2 \underbrace{\xrightarrow{a/0}}_{w=a} s_2 \underbrace{\xrightarrow{a/0}}_{h} s_2 \underbrace{\xrightarrow{a/0}}_{x=a} s_2 \underbrace{\xrightarrow{a/0}}_{w=a} s_2 \underbrace{\xrightarrow{a/0}}_{h} s_2 \underbrace{\xrightarrow{b/0}}_{x=b} s_3 \underbrace{\xrightarrow{a/1}}_{w=a} s_1$$

A state $p_1 = H(0)$ with characterization $\{a \mapsto 0\}$ is learned, alongside its transitions ‘a/0’ (a loop) and ‘b/0’ (leads to state p_2). The state p_2 with characterization $\{a \mapsto 1\}$ is also learned so that we can start learning transitions from p_2.

$$s_1 \underbrace{\xrightarrow{a/0}}_{h} s_2 \underbrace{\xrightarrow{b/0}}_{\alpha=b} s_3 \underbrace{\xrightarrow{a/1}}_{x=a} s_1 \underbrace{\xrightarrow{a/0}}_{w=a} s_2 \underbrace{\xrightarrow{a/0}}_{h} s_2 \underbrace{\xrightarrow{b/0}}_{\alpha=b} s_3 \underbrace{\xrightarrow{b/0}}_{x=b} s_1 \underbrace{\xrightarrow{a/0}}_{w=a} s_2$$

After this subinference of hW-inference, we have learned the conjecture in Figure 10. Since we learned the transition ‘b/0’ from state p_2 to p_1 and loop transition with ‘a/0’ on p_1, we are in the state s_2 of the conjecture. As the learned conjecture is complete, we can ask the oracle for a counterexample. Assuming the provided counterexample is the following:

$$s_2 \xrightarrow{b/0} s_3 \xrightarrow{a/1} s_1 \xrightarrow{b/1} s_1$$

We identify a new W-inconsistency since we observed ‘b/0’ from state p_1. In that case, W is extended by ‘b’, so that $W = \{a, b\}$, which is characterizing (see the W-set in Table 3). Therefore, we can find the correct conjecture in the next subinference by following the same idea.

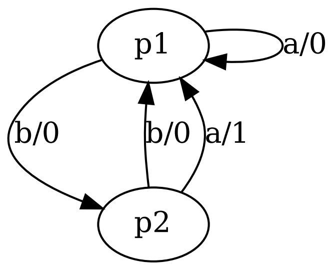

Figure 10 – Conjecture after the second subinference of hW-inference.

Source: Adapted from Halm (2021).

4.3 Learning Separating Sequences from Traces

We present a method for computing a set of separating sequences from traces to compose a characterization set which is then used in hW-inference. The main idea is that the trace is analyzed to identify suitable candidates for W.

The red lines (3, 4, 22, and 23) of Algorithm 1 indicate the main points where the proposed method is applied in hW-inference. These steps refer to the same procedure but represent different learning stages.

Before the first **repeat** loop (line 5), during the initialization phase, our procedure is called to provide a tentative initial W-set for hW-inference. The separating sequences inference method, called W-inference, produces a set W' of input sequences and requires three parameters (T,k,n) where:

- T is the set of traces that will be used as a source of information for learning separating sequences. During the initialization phase, the only member of T is the provided trace ω. Afterward, T will be composed of both ω and the execution trace,
- k is the maximum length of the elements of W', and
- n is the desired number of elements in W'.

It is worth mentioning that the only assumption on the traces of T is that they have to be valid examples of the system behavior. We do not assume the traces start from a determined state of the machine (e.g., the initial state), and the traces do not have to be in order.

Algorithm 2 presents the W-inference method. The method is described in the remainder of this section through an example.

The inference method starts by obtaining k-size subsequences of the provided input/output trace ω for each of its indexes, e.g., considering $\omega =$ 'a/0.b/1.b/2.c/0' and $k=2$ we have '{a/0.b/1, b/1.b/2, b/2.c/0, c/0}' as its subsequences. Therefore, the number of subsequences equals the number of input/output pairs in ω, and each subsequence has k pairs, except the $(k-1)$-last pairs. Since we only look k-times ahead for each index, obtaining the subsequences of ω is done in $O(kn)$, where $n = |\omega|$. For illustration, Figure 11 presents an extract of trace ω and its 12 subsequences for $k=4$. Obtaining the subsequences is indicated in line 4 of Algorithm 2, where each subsequence is represented as a tuple $(s^i, s^o, \overrightarrow{s})$. s^i is the input sequence, s^o is the output sequence, and $\overrightarrow{s}$ is the next input (the immediate following input seen after the input/output sequence, which can be ε if there is no next input).

Once the subsequences of ω are available, we start filling a tree named W-tree that aggregates and explores the information provided by each subsequence. When a

ω = ... b/0 a/4 b/0 c/0 a/0 c/1 b/2 c/1 b/0 b/0 b/1 b/0 ... $k = 4$

ω_1 = b/0 a/4 b/0 c/0	ω_5 = a/0 c/1 b/2 c/1	ω_9 = b/0 b/0 b/1 b/0
ω_2 = a/4 b/0 c/0 a/0	ω_6 = c/1 b/2 c/1 b/0	ω_{10} = b/0 b/1 b/0
ω_3 = b/0 c/0 a/0 c/1	ω_7 = b/2 c/1 b/0 b/0	ω_{11} = b/1 b/0
ω_4 = c/0 a/0 c/1 b/2	ω_8 = c/1 b/0 b/0 b/1	ω_{12} = b/0

Figure 11 – An extract of trace ω and its subsequences for $k = 4$.

Source: Elaborated by the author.

visual representation of the W-tree is needed, one could use the following notation for its components.

- ⊚ is the root of the tree,
- ◯ is an inner node attached to dotted and dashed boxes,
- ▽ is a leaf node, i.e., all output sequences listed in its dashed box are terminal,
- $\rightarrow$ indicates an edge of the tree whose label corresponds to an input from a subsequence of ω,
- A dashed box, named o-box as a short of output-box, indicates the input sequence w', starting from the tree's root, that leads to the node attached to it. The o-box also maps each output sequence observed in ω for w' to the immediate next input observed in ω after the end of w',
- A dotted box, named r-box as a short of response-box, indicates the number of states the input sequence that leads to its attached node is capable of distinguishing.

It is worth remarking that all types of nodes (⊚, ◯, ▽) in the tree are numbered. The node number indicates the order in which it is inserted in the tree.

Filling the W-tree is performed in a top-down strategy by inserting each subsequence of ω into the tree from its root so that each input becomes an edge of the tree, connecting the previous input to the current one. As the W-inference method is proposed in the context of deterministic automata, the W-tree must also be deterministic so there is no more than one outbound edge with the same input for a given node. Figure 12 presents the resulting W-tree from inserting all 12 subsequences indicated in Figure 11. It is worth highlighting that, as we insert k-size sequences in the tree from its root node, the depth of the resulting W-tree is bound to the value of k. This process is indicated in line 6 of Algorithm 2.

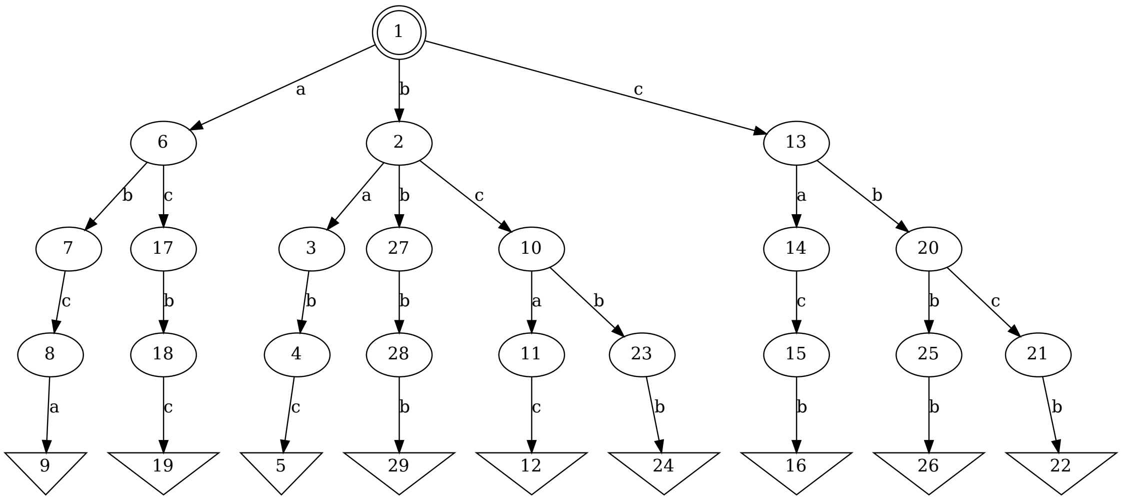

Figure 12 – W-tree with only nodes and edges for the extract trace from Figure 11.

Source: Elaborated by the author.

As we insert the subsequences, the o-boxes are filled with each output sequence observed in ω for the input sequence inserted in the tree. Each o-box states, in its first line, the input sequence that leads to its corresponding node. The following lines of the o-box map the observed output sequences to a set of immediate next inputs. Figure 13 shows the resulting W-tree after the creation of the o-boxes for the example in Figure 12.

Using node 2 as an example, we can describe the filling of the o-boxes. Node 2 represents the input sequence ‘b’, so, at first, its o-box has only the header line ‘b/’. As the subsequence ω_1 is inserted in the tree, we also insert the output sequence ‘0’ into the o-box. As the next input of first ‘b/0’ in ω_1 is ‘a’, the o-box is updated with the mapping ‘$0 \mapsto \{a\}$’. As we insert the subsequence ω_3, we notice that a mapping from ‘0’ already exists, and since the next input of ‘b/0’ is ‘c’, we add the input to the mapping, resulting in ‘$0 \mapsto \{a,c\}$’. A new entry is added to the o-box as we insert ω_7 since the output for the first ‘b’ of the subsequence is ‘2’ instead of ‘0’. In that case, the insert ‘$2 \mapsto \{c\}$’ into the o-box. Following this logic, ω_9 updates the mapping from ‘0’ adding the input ‘b’, so that the resulting map is ‘$0 \mapsto \{a,c,b\}$’. The subsequence ω_{10} does not modify the entries of the o-box since a mapping from ‘0’ to ‘b’ already exists. ω_{11} forces the addition of a new mapping as ‘$1 \mapsto \{b\}$’. Finally, ω_{12} updates the mapping from ‘0’ with the terminal symbol ε, indicating that no input was observed after the current input in the sequence, resulting in ‘$0 \mapsto \{a,c,b,\varepsilon\}$’. This idea is valid for constructing all the other o-boxes in the W-tree. The filling of the o-box of node 2 is illustrated in Figure 14 as a relation between

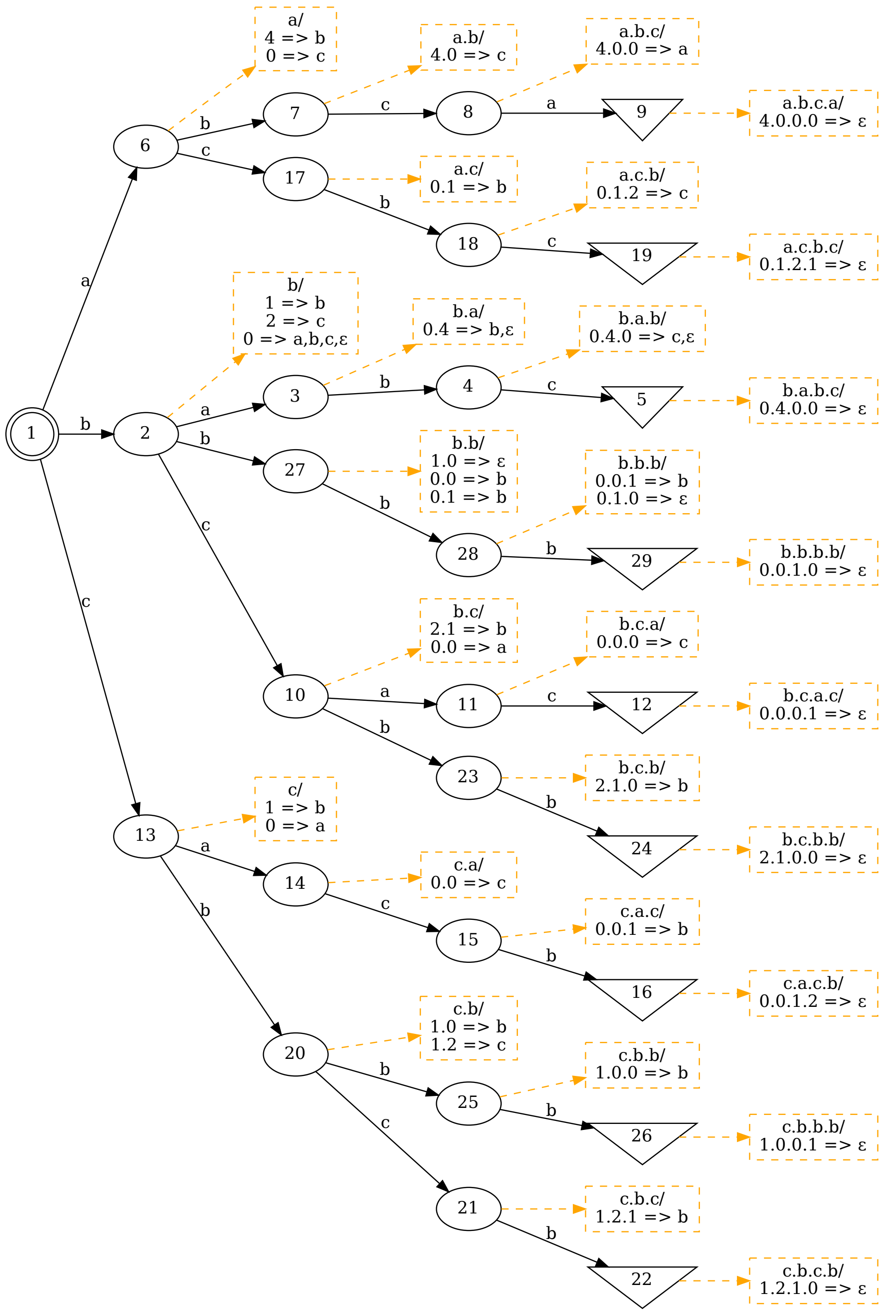

Figure 13 – W-tree with o-boxes for the extract trace from Figure 11.

Source: Elaborated by the author.

the elements of $\boldsymbol{\omega}$ and the elements of the o-box.

Filling the o-boxes of the W-tree corresponds to the line 6 of Algorithm 2. This is the first line where the notation *object*[*property*] is used, which means "get *property* of *object*". In this line, for example, node[o-box] means get the corresponding o-box of *node*.

After all subsequences of $\boldsymbol{\omega}$ are in the tree, and all the o-boxes are filled, we fill the r-boxes, following a bottom-up strategy from the leaf nodes to the tree's root (see lines

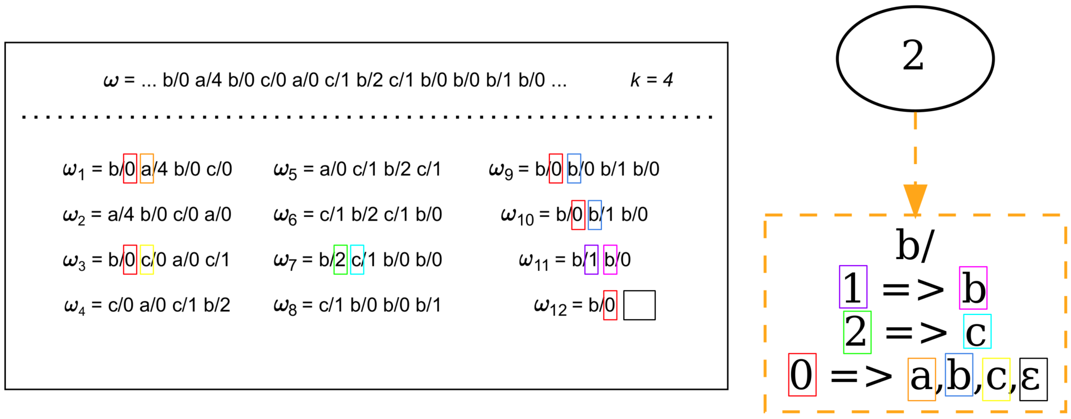
ω = ... b/0 a/4 b/0 c/0 a/0 c/1 b/2 c/1 b/0 b/0 b/1 b/0 ...
k = 4
ω_1 = b/0 a/4 b/0 c/0
ω_2 = a/4 b/0 c/0 a/0
ω_3 = b/0 c/0 a/0 c/1
ω_4 = c/0 a/0 c/1 b/2
ω_5 = a/0 c/1 b/2 c/1
ω_6 = c/1 b/2 c/1 b/0
ω_7 = b/2 c/1 b/0 b/0
ω_8 = c/1 b/0 b/0 b/1
ω_9 = b/0 b/0 b/1 b/0
ω_{10} = b/0 b/1 b/0
ω_{11} = b/1 b/0
ω_{12} = b/0
2
b/
1 => b
2 => c
0 => a,b,c,ε

Figure 14 – Relation between the elements of ω and the o-box of node 2 from Figure 13.

Source: Elaborated by the author.

10 to 27 of Algorithm 2). We consider the number r of different responses to an input sequence as the number of states it is capable of distinguishing. It is essential to notice that this number can not be seen as a final or exact measure since r is obtained from the trace instead of the FSM conjecture. Consequently, we can describe r as a potential value for the number of states in the FSM that a given input sequence would distinguish.

Each r is calculated by obtaining the child with the maximum number of responses r' among the node's children and adding its r' to the number of output sequences that do not lead to the chosen child. Figure 15 presents the complete W-tree for the example from Figure 11. As an example of the r calculation method, consider node 9. As a leaf node, node 9 does not have children. In that case, r equals the number of different output sequences in the node's o-box; so $r = 1$. This simple equivalence is valid for all leaves in the W-tree of the example.

When a node has only one child, we calculate its r by summing its child's r with the number of different output sequences in its o-box that do not lead to its child. For example, observe node 28, which has only child 29, with $r = 1$. The input that leads to node 29 from node 28 is 'b'; by verifying its o-box, we can notice that the only output sequence that does not map to 'b' is '0.1.0'. In that case, the r value of node 28 equals 2 (one from its child node and one from its o-box). A similar scenario is seen in node 15, for example. Node 15 has only one child, but only one entry in its o-box; then, its r equals its child's r, $r = 1$.

Node 2, for instance, has more than one child. In scenarios like this, we need to identify the child with the highest r value and then identify the output sequences that do not lead to that child. The children of node 2 are node 3 ($r = 1$), node 27 ($r = 3$), and node 10 ($r = 2$); then, node 27 is the chosen child since it has the highest r value. In this case, there is no space for doubt as all r values are different. For a node like 6, a choice by other criteria must be made since both its children have $r = 1$. Tiebreaker criteria could include lexicographic order, leftmost and rightmost elements, random selection, and others if needed. In our implementation of the W-inference method, we opted for the order of the insertion of the nodes by their number, so the chosen child for node 6 would be node 7 since it was inserted before node 17.

As we have the “best” child (node 27) for node 2, we need to identify the number of different output sequences that do not lead to node 27; it means all the output sequences that do not map to the input ‘b’. Node two’s o-box indicates that both output sequences ‘1’ and ‘0’ lead to the input ‘b’, so the only output sequence that does not map to ‘b’ is ‘2’, which leads to node ten by the input ‘c’. Finally, node two’s r equals to its “best” child’s r ($r = 3$ of node 27) plus node ten’s $r = 2$; so node two’s $r = 5$.

There is one exception to this rule. Observe the r value for node 27; following the presented rule, its r value would be two instead of three. However, we also must consider the output sequences that map to the terminal symbol ε. Thus, the r value for node 27 is three (two from its “best” child plus one from the output ‘1.0’ that maps to ε).

After calculating all r-boxes, the W-tree is complete, and we can build a set of tentative separating sequences from it. To do so, we search in the tree, starting from its root, the shortest n-sequences that provided the highest values of r (see lines 29 to 44 of Algorithm 2). In the example, if $n = 1$, the longest “best” input sequence is ‘b.b.b.b’ since we start from the tree’s root, this input sequence traverses the nodes with the highest values for r. However, as we aim to take the shortest sequence, we must follow the following rule to decide whether to go deeper into the tree or stop.

- If a node has no children, stop. This will be the longest “best” sequence.
- If a node has only one child, verify if its child has a higher value for r than its father; if so, proceed with the child node; otherwise, stop.
- If a node has more than one child, proceed with the child with the highest r value. If children have the same r value, we could apply the same strategy to identify the “best” child node for calculating r.

This idea allows us to select the first two inputs from ‘b.b.b.b’ since ‘b’ represents node 2, which has three children; we chose node 27 as its “best” child. Since ‘b.b’ represents node 27 and its only child (node 28) does not have a higher r value than node 27, we can take ‘b.b’ as the shortest “best” input sequence for $n = 1$. If $n = 2$, we would select ‘b.b’ as the first member of the W and either ‘b.c.a’ or ‘b.c.b’ as the second because the nodes 11 and 23 have the same r value. Node 11 would be selected in our implementation so that the input sequence would be ‘b.c.a’. When $i = 1$, the function on line 30 of Algorithm 2 always returns the root node, but for the subsequent values of i, it is necessary to choose a different node that will make W diverge; otherwise, the same input sequence would be selected for all n values.

It is essential to notice that as the number of desired sequences n increases, the confidence level in the "quality" of the W-set decreases since there is a risk that the selected sequences may overlap with each other and cause redundancy. Therefore, $n = 1$ is safer and enough to improve the inference of the machine (see Section 5.3).

As soon as the learned set is available, we can merge it with the current W of hW-inference (see lines 4 and 23 of Algorithm 1). In the first inference, an initial W will be learned, and, as there is no conjecture yet, no W-set reduction phase is executed. During the subsequent inferences (see line 22 of Algorithm 1), we perform the reduction phase immediately after improving W to benefit from removing redundant sequences and possibly shortening the remaining ones.

Although we described the stages of the proposed method as consecutive activities, we implemented them slightly differently for better performance. We can compute the k-size subsequences from the provided trace, insert them in the W-tree, and simultaneously fill the corresponding o-boxes. Besides, we implemented filling the r-boxes of the tree as a separated phase, but it could also be done in parallel to inserting sequences in the tree.

4.4 W-inference, hW-inference, and W-set reduction

Although our proposed method is related to hW-inference and mutually benefits each other, they do not need to be strictly bound. We designed and implemented W-inference as an improvement for hW-inference, making it more efficient and bringing an initial attempt to make it a hybrid learning algorithm. Still, its benefits can be exploited by other algorithms. W-inference is self-sufficient and can be used by different model learning algorithms with little or no changes to its idea.

W-inference mainly benefits hW-inference in two different points:

- **W-inference provides an tentative initial W-set for hW-inference**. When W-inference is not used, hW-inference needs to build its initial W by interacting with the SUL, which implies a cost that can be minimized by W-inference.

- **W-inference provides a new source of information to improve W**. During the execution of hW-inference, without W-inference, W is only revised when an inconsistency is found, or while processing a counterexample the oracle provides. With W-inference, hW-inference's execution trace is analyzed together with the provided trace to find new tentative sequences to improve W after each sub-inference of hW-inference.

In turn, hW-inference mainly benefits W-inference by providing a data source for computing the separating sequences since its growing execution trace is available for W-inference.

Another interesting aspect of the interaction between W-inference and hW-inference is evident when W-set reduction is available. As the W-inference method does not know the current W and its sequences, the W-set reduction phase of hW-inference provides an additional benefit as it checks if the newly added sequences improve the current level of characterization or remove them, if necessary; without this feature, W-inference could provide already insert sequences or sequences that do not improve the characterization level of the set.

4.5 Final Considerations

In this chapter, we introduced the proposed method of this dissertation, called W-inference. The method was detailed, and all its structures and stages were illustrated through examples. Although W-inference can be applied to different learning algorithms, we designed it to fit into hW-inference, which leads to a favorable relation between them, so this approach was also discussed in this chapter.

Algorithm 2 – W-inference algorithm

```
procedure WINF($T, k, n$)
    root $\leftarrow$ createNode()
    for each $t \in T$ do
        $S \leftarrow k$-subsequences($t, k$)
        for each $(s^i, s^o, \overrightarrow{s}) \in S$ do
            node $\leftarrow$ insert(root, $s^i$)
            updateMapping(node[$o$-box], $s^o$, $\overrightarrow{s}$)
        end for
    end for
    $L \leftarrow$ getAllLeaves(root)
    for each $l \in L$ do
        node $\leftarrow l$
        $r \leftarrow 0$
        while node $\neq$ root do
            if node $= l$ then
                $r \leftarrow$ getNumberOfResponses(node[$o$-box])
            else
                $\star$child $\leftarrow$ getBestChild(node)
                children $\leftarrow$ getNonOverlappingChildren(node)
                $r^{\varepsilon} \leftarrow$ getNumberOfEpsilonInputs(node[obox])
                $r^c \leftarrow \Sigma_{i=0}^{|\text{children}|-1}$ (children$_i$[$r$-box][$r$])
                $r = \star$child[$r$-box][$r$] $+ r^c + r^{\varepsilon}$
            end if
            node[$r$-box][$r$] $\leftarrow r$
            node $\leftarrow$ getParentNode(node)
        end while
    end for
    $W \leftarrow \emptyset$
    for $i \leftarrow 1$ to $n$ do
        node $\leftarrow$ getRootOr$W$DivergenceNode(root, $W$)
        $w \leftarrow \varepsilon$
        while $w = \varepsilon$ do
            if node $\in L$ then
                $w \leftarrow$ node[$o$-box][input sequence]
            else if getNumberOfTransitions(node) $> 1$ then
                node $\leftarrow$ getBestChild(node)
            else if getFirstChild(node)[$r$-box][$r$] $\leq$ node[$r$-box][$r$] then
                $w \leftarrow$ node[$o$-box][input sequence]
            else
                node $\leftarrow$ getFirstChild(node)
            end if
        end while
        $W \leftarrow W \cup \{w\}$
    end for
    return $W$
end procedure
```

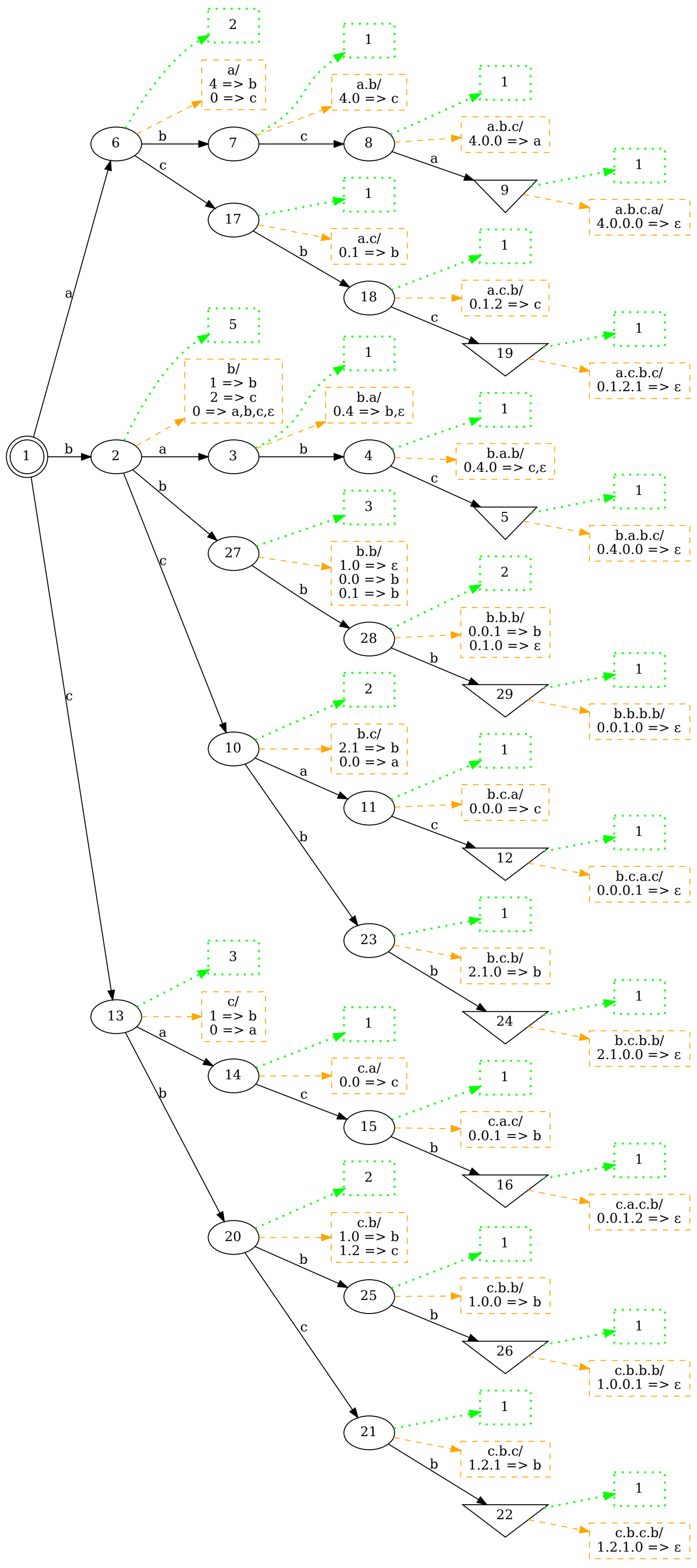

Figure 15 – Complete W-tree from the extract trace from Figure 11.

Source: Elaborated by the author.

CHAPTER

5

EXPERIMENTAL EVALUATION

5.1 Initial Considerations

The previous chapter introduced and contextualized the proposed method, called W-inference, its structure, and its relation to hW-inference. In this chapter, we aim to assess the impact and benefits of the proposed method in the context of FSM inference. To do so, we implemented W-inference in the same ecosystem of hW-inference and performed a case study.

This chapter is organized as follows. Section 5.2 details the case study performed and presents its outcomes. Section 5.3 discusses the findings of the case study.

5.2 Case Study

In this section, we describe the experiment conducted to assess the impact of the proposed method on hW-inference. It is essential to notice that we used 40 traces in the experiment, each of which had a random seed (the oracle uses random seeds during random walks to check the conjecture). The traces were generated by random walking on the original FSM, starting from random states. All traces have the same length of 1.00E+06 input/output pairs. To assess the impact of the size of the provided traces, we progressively increased the length from 1.00E+03 pairs to 1.00E+06 pairs.

We first augmented the implementation of hW-inference in SIMPA[1]. SIMPA is a recursive acronym for "SIMPA Infers Model Pretty Automatically" introduced by Büchler *et al.* (2014) and has been used in different projects since it is a model inference framework that provides implementations for various algorithms, including hW-inference (GROZ *et*

[1] SIMPA is available at <http://vasco.imag.fr/tools/SIMPA> and <https://gricad-gitlab.univ-grenoble-alpes.fr/SIMPA/SIMPA>.

al., 2020).

We used our implementation in SIMPA on a case study called "*Scanette*" first used in (UTTING *et al.*, 2020) that has been used in different learning approaches, including hW-inference. *Scanette* represents a supermarket self-service scanning system. The complete description of *Scanette* is available in a technical report of (PHILAE, 2019). Besides that, a driver is included in SIMPA, and a game-like simulation is avaliable[2].

As a complex non-deterministic reactive system, we used an adapted deterministic abstraction of *Scanette*, also used by Halm *et al.* (2022). The deterministic *Scanette* can be described as a DFSM with 121 states, 15 inputs, and more than 1800 transitions.

We used the length of the execution trace as the critical measure of impact, as hW-inference is an active inference algorithm. Halm *et al.* (2022) indicate that the average trace length of basic hW-inference, without any W-set reductions or improvements by the method proposed in this dissertation, is around 3.09E+06 inputs, with a standard deviation of 2.66E+06 inputs. Besides, when only the W-set reduction method is used in hW-inference, we can obtain an average trace length of 2.40E+06 inputs, with a standard deviation of 1.92E+06, which indicates an improvement of around 22%.

Before running the experiment on the W-inference method, we performed a sanity check on randomly composed W-sets since it was possible that providing any set of input sequences could improve the FSM learning algorithm. We randomly picked inputs from the input alphabet of *Scanette* 35 times and executed the basic hW-inference providing them as initial W-sets. We observed an average trace length of 3.99E+06 inputs, with a standard deviation of 3.04E+06. This simple test confirms that we can not assume that any provided W-set can improve the learning algorithm, contrasting with the proposed method.

To understand the impact of the amount of data and the value of k provided to W-inference, we executed it for progressively bigger subtraces (1.0E+03, 1.0E+04, 1.0E+05, and 1.0E+06) of the 40 randomly generated ones and k varying from 1 to 5. Table 4 summarizes the results per provided trace length and k.

Figure 16 shows the relation between the length of the provided trace and the length of the execution trace, produced by hW-inference due to interacting with the SUL so that we can compare the average results by k value.

We also calculated the maximum number of sequences in the W-set per iteration of hW-inference augmented with our proposed method. As we executed the algorithm 40 times per trace file and k value, we obtained 800 values for the number of sequences in W. All observed values are in Figure 17. The figure suggests that we have more dispersed values for $|W|$ as the trace length increases, but the number of sequences in W is closer to

[2] JavaScript implementation of *Scanette*: <https://fdadeau.github.io/scanette/>

Table 4 – Results per trace length and k value.

Trace Length	k	Avg. Execution Trace Length	Standard Deviation	Avg. Size of W	Avg. Length of $w \in W$
1.00E+03	1	2.49E+06	1.30E+06	26	3
	2	2.21E+06	1.18E+06	24	4
	3	2.58E+06	1.83E+06	29	4
	4	2.83E+06	1.56E+06	29	4
	5	3.36E+06	1.95E+06	31	4
1.00E+04	1	2.42E+06	6.24E+05	22	3
	2	1.19E+06	6.21E+05	25	4
	3	2.10E+06	1.07E+06	23	3
	4	3.13E+06	1.63E+06	27	4
	5	2.96E+06	1.80E+06	28	4
1.00E+05	1	2.14E+06	5.41E+05	23	3
	2	1.68E+06	1.08E+06	25	4
	3	2.36E+06	1.16E+06	23	3
	4	2.44E+06	1.48E+06	24	4
	5	2.60E+06	1.51E+06	24	3
1.00E+06	1	2.42E+06	8.14E+05	23	3
	2	1.31E+06	6.74E+05	26	4
	3	2.04E+06	1.09E+06	23	3
	4	2.76E+06	2.13E+06	25	3
	5	2.25E+06	1.12E+06	25	3

Source: Elaborated by the author.

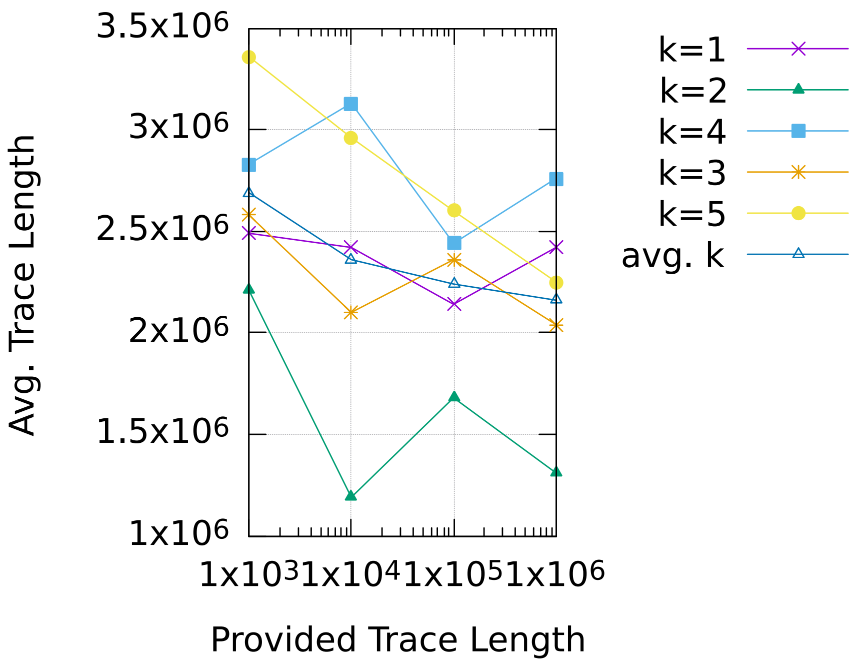

Figure 16 – Relation between the provided trace length and execution trace length per k value.

Source: Elaborated by the author.

the average (25 sequences) for shorter traces.

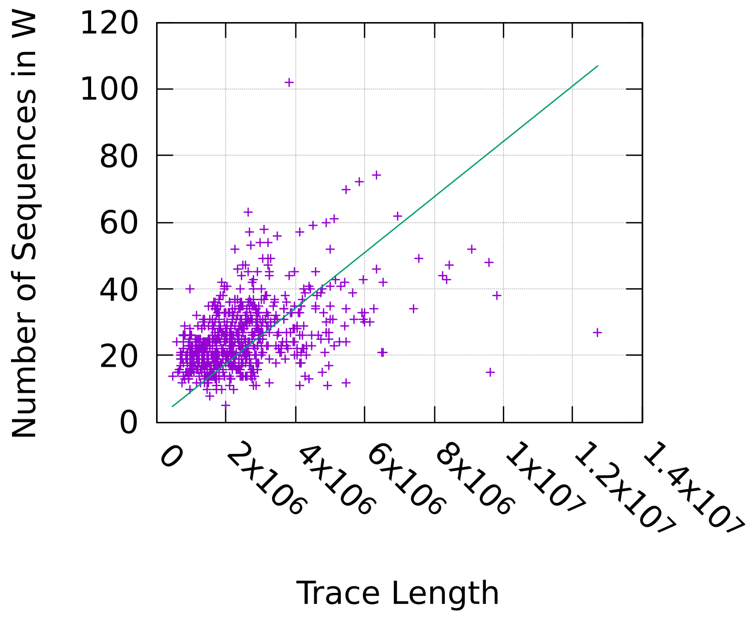

Figure 17 – Relation between the execution trace length and the length of W.

Source: Elaborated by the author.

5.3 Discussion

As highlighted in Figure 16, the parameter k and the length of the provided trace ω impact the size of the execution trace. The combination of these parameters can either improve or worsen the learning process. However, it is essential to note that another crucial factor impacts the results: the "quality" of the provided trace. As the proposed method depends on a given trace, there is no guarantee that the trace reflects all essential aspects of the behavior of the original FSM. Therefore, it is possible to mislead the inference to produce a non-effective W-set.

Even though the cases ($k = 5$ and $|\omega| =$ 1.0E+03) and ($k = 4$ and $|\omega| =$ 1.0E+04) are worse than the average basic hW-inference, all other combinations of parameters that we observed in our experimental evaluation produced better results, which are capable of improving the learning process. The mean value of all observed sizes of execution traces is around 2.36E+06, which indicates an improvement of around 24% compared to basic hW-inference. If we calculate the improvement by k value, we can highlight that $k = 2$ produced the best results as the improvement, in mean, was around 48% compared to basic hW-inference and 33% when compared to the version of hW-inference with W-set reduction. Table 5 summarizes the key results for the methods, on average, for the execution trace length. The values in brackets represent the difference between the result and basic hW-inference.

Table 5 – Key results for W-inference, hW-inference, and W-set reduction.

Method	**Avg. Execution Trace Length**	**Standard Deviation**
Basic hW-inference	3.09E+06	2.66E+06
W-set Reduction hW-inference	2.40E+06 *(-22.33%)*	1.92E+06 *(-27.82%)*
W-inference hW-inference (Avg.)	2.36E+06 *(-23.63%)*	1.43E+06 *(-46.24%)*
W-inference hW-inference (k=2)	1.60E+06 *(-48.22%)*	8.89E+05 *(-66.58%)*

Source: Elaborated by the author.

As shown in Figure 16, as we increase k, the execution trace length also increases, except for $k = 1$. This scenario was expected since $k = 1$ produces a W-tree with depth one and generally does not provide enough information to compose the W-set from the learned separating sequences. For *Scanette*, $k = 2$ was the best option. However, it is not the universally best choice since the case study, though based on a real system and used by different research projects, does not reflect all different types of systems and their characteristics. More experiments would be necessary to assess better the best combination of parameters, which could be done with machines from benchmarks.

As we proposed our version of hW-inference as an augmented version of the one implemented by Halm *et al.* (2022), it is crucial to verify the impact of our changes in W on the key metrics to W-set reduction: number of sequences in W and the length of its sequences. As we can observe in Figure 17, most values of $|W|$ are smaller than 40 input sequences, which fits in the same scenario as hW-inference with W-set reduction. Besides that, Table 4 indicates the average length of sequences in W by trace length and k; we can notice that the mean value is 4, and the minimum and maximum values are 1 and 7, respectively, indicating an improvement since the values for basic hW-inference without W-inference are around 9 inputs in mean, and hW-inference with only W-set reduction has a mean value of 8 inputs. Figure 18 shows a counting on how many times the mean length of each sequence in W was identified in the 800 executions of the experiment, which most fit between 3 and 4.

Another important finding, exposed in Table 5, is the stability of the proposed method. We noticed that the augmented versions of hW-inference reduce the standard deviation σ of their respective execution traces. The basic hW-inference has a high σ value of around 2.66E+06 inputs and hW-inference with W-set reduction lower σ to 1.92E+06, which indicates less dispersion of the results from the mean value. When our proposed version of hW-inference is used, we observe a standard deviation of 1.43E+06, on average, and an even lower σ (8.89E+05) when the parameter $k = 2$ is used.

Lastly, another important remark concerns the length of the provided trace. As illustrated in Figure 16 by the relation between the length of the provided trace and its corresponding execution trace length, especially for the average of all k values, there is an improvement when larger traces are provided; still, we may not consider it worth

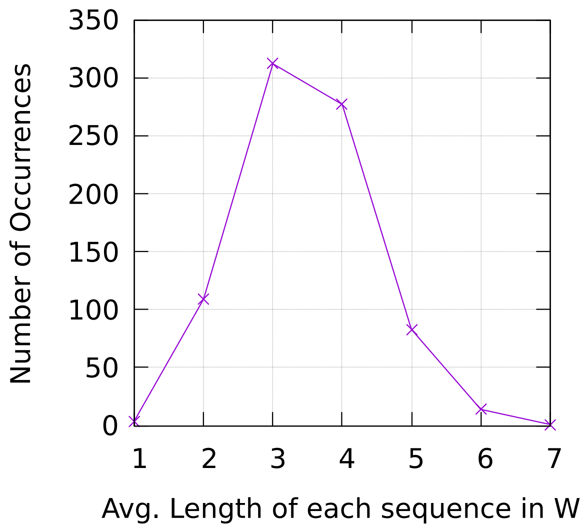

Figure 18 – Number of occurrences of the lengths of $w \in W$.

Source: Elaborated by the author.

since W-inference was able to provide similar results with shorter traces. For example, the average execution trace length for traces of size 1.0E+04 is 2.36E+06, and for traces of size 1.0E+06, it is 2.16E+06, which indicates an improvement of around 8.50%, but it was necessary and increase of 100 times in the trace size. Those close results suggest that W-inference can produce "good" results with shorter traces, which is an advantage since it is not necessary to collect and process large quantities of data.

5.4 Final Considerations

In this chapter, we described the experimental assessment of our proposed method, W-inference, through a case study called *Scanette*. The key findings were discussed and contextualized. As a key outcome, we can highlight that our implementation of hW-inference with W-inference was capable of providing better results (improvements up to 48%) than its previous versions (basic hW-inference and hW-inference with W-set reduction).

CHAPTER 6

CONCLUSION

As indicated in Chapter 2, software models, also named computation models, have been applied in Software Engineering for decades for software specification and testing. Models, especially those capable of representing the behavior of systems, have become fundamental concepts for developing and maintaining software systems as they serve as a basis for automating numerous approaches, such as testing, verification, validation, and refinement of systems, including the well-known Model-Based Testing (a branch of software testing that considers processes and techniques for automatically deriving test cases from models), discussed in Section 2.2.3.

Section 2.2.2 discusses the duality of the benefits of models and their costs. Although models are important in Software Engineering and base the automation of many testing and quality assurance activities, it is often difficult to find them in practice. The main obstacle to using models in practice is their non-trivial creation and maintenance, besides being unrealistic to expect developers to maintain models of systems, especially in an agile software development context where both requirements and implementation is continuously changing.

One option to mitigate this problem is model inference which provides the possibility to automatically, or at least with little human interaction, learn a model that represents the behavior of a system. This process can be performed directly on the system's source code or considering it as an executable black box and, therefore, without access to its internal structure. In this context, Section 2.2.2 and Section 3.3 detail the two main approaches to model inference: *passive inference* (builds models from examples of the behavior of a system) and *active inference* (builds models from interacting with the system).

As our research focus, this dissertation highlights the Finite State Machine, discussed in Section 2.2.1.1 and Chapter 3. Finite State Machines are a model type that aims to represent behavior through states and transitions. This model type has been used for decades to specify and test computer systems. One aspect that has gained increasing attention is the inference of FSMs, which can discover a Finite State Machine from interacting with a system or a finite set of previous observations of its execution. In Section 3.3, we present and contextualize several algorithms for active and passive learning of FSMs, including the active learning algorithm hW-inference, the main inference algorithm of this dissertation that is described in Section 4.2.

In Chapter 4, we detail the main contribution of this dissertation: a method for learning separating sequences from traces and applying it to improve the process of model inference. A separating sequence is an input sequence capable of distinguishing a pair of states of a machine by yielding different output sequences for each state. Our proposed method works by taking a set of traces and processing them to extract all their k-length subsequences and using them to build a data structure called W-tree that summarizes the relevant observations of the system's behavior indicated in the traces. The method's output is a set of the n-best separating sequences. The model inference algorithm then uses this output to improve its W-set (characterization set is a set of separating sequences that can distinguish all distinct pairs of states in the FSM). The complete description and exemplification of the proposed method are in Section 4.3. The integration of the proposed method with the model learning algorithm is available in Section 4.2 and Section 4.4.

Finally, Chapter 5 describes the experimental evaluation of the proposed method, besides discussing its outcomes. To assess the impact of the proposed method on the model inference process, we implemented the proposed method in the same ecosystem of hW-inference, called SIMPA. We performed a case study called *Scanette*, which represents a deterministic FSM model of a supermarket self-service scanning system with 121 states, 15 inputs, and more than 1800 transitions. In the experiment, we used 40 different traces and five different values for the parameter k, so that 800 executions of the model inference method were performed. As key findings, we can indicate that the proposed method can indeed improve the learning process since we observed an average improvement of 24% compared to the basic version of hW-inference and an average gain of 48% when the parameter $k = 2$ was used i.e., with a sequence with length 2, as discussed in Section 5.3.

In Section 1.2, we listed the main objectives of this dissertation so that we can verify if such objectives were accomplished.

1. **Describe and implement a method for learning separating sequences**. As discussed in Section 4.3, we designed a method for learning separating sequences from traces, called W-inference, and implemented it in SIMPA, an open-source project.

2. **Integrate the proposed method into a *active learning* Finite State Ma-**

chine inference algorithm. We designed the proposed method to be integrated into a model learning algorithm called hW-inference so that they both provide benefits for each other. We discussed this relation in Section 4.4.

3. **Use passive learning to improve *active learning***. As Section 2.2.2 indicates, *hybrid learning* is a well-known term in model inference to specify algorithms that use active and passive learning together. Before our proposed method was developed and integrated into SIMPA, hW-inference was an *active learning* algorithm; now, we introduced the first steps to make hW-inference a hybrid algorithm.

4. **Prepare and conduct a case study to assess the proposed method's impacts on the model inference process**. We conducted a case study based on a real software model used by other research projects. We used 40 different traces and performed the inference process 800 times to assess the impact of the traces and parameters in the inference. As a result of the case study, we can highlight that when hW-inference is integrated W-inference, we can achieve improvements up to 48%. Chapter 5 is dedicated to presenting the experimental evaluation of the proposed method.

6.1 Limitations

It is worth highlighting some limitations of the proposed method. Although the experimental evaluation indicated that a significant improvement could be achieved by selecting one separating sequence per execution of the proposed method, our implementation of the proposed method and the experimental evaluation is limited to parameter $n = 1$ (see Section 5.2). However, if a proper way to select new candidates without redundancy or interpolation were proposed, there is room for even better results. Also, our proposed method was implemented to receive one given trace and selects the separating sequences from both the given trace and the current execution trace of the learning algorithm. Still, it could be extended to receive multiple traces as input since the method was designed to handle a set of traces. However, it would be interesting to perform experiments on this aspect since our experimental evaluation revealed that, at least for *Scanette*, providing large traces is not necessarily worth it, and perhaps it would be a similar case for providing more traces.

Besides that, another limitation or assumption is that our proposed method is suitable for deterministic FSM inference algorithms but was originally designed and implemented for the hW-inference active learning algorithm, so it is still necessary to experiment better with its integration with other learning algorithms.

The proposed method also considers only the set of provided traces and its parameters k,n to compute the separating sequences, which states the number of states that selected separating sequences can distinguish as potential values instead of concrete ones. A more precise distinguish rate could be achieved if the proposed method were adapted to consider the current conjecture to check the separating sequences.

6.2 Future Work

As future work, we can indicate the need to assess the proposed method on different systems than the one used in our case study, which could be done using benchmarks such as the benchmarks collected at the Radboud[1] university (NEIDER *et al.*, 2019). Besides, the research conducted in this dissertation could be extended to mitigate its limitations, so augmenting the proposed method and its implementation to properly handle a set of provided traces and $n > 1$ and using the current conjecture to verify the selected separating sequences would be a starting point for future work. Also, it would be interesting to apply the proposed method to other learning algorithms since verifying its impact only in hW-inference does not provide a more general view of its impact on learning algorithms. The proposed method only considers the preset version of hW-inference. Still, one could adapt the proposed method and its implementation to be suitable for the adaptive version of hW-inference as well, which applies a tree-like structure as the representation of its W-set and homing sequences, that provides similarities to the W-tree structure used by the proposed method. However, the W-set reduction method of hW-inference would also need to be adapted since it is only suitable for the preset version. Finally, investigate which other parts of the hW-inference learning process could benefit from the *passive learning* data introduced by our proposed method; a potential candidate would be checking for counterexamples in the provided trace to reduce the costs of asking the oracle.

[1] <http://automata.cs.ru.nl/Overview#Mealybenchmarks.>

BIBLIOGRAPHY

AHO, A. **Compilers : principles, techniques, & tools**. Boston: Pearson/Addison Wesley, 2007. ISBN 0321486811. Citation on page 42.

ANGLUIN, D. Learning regular sets from queries and counterexamples. **Information and Computation**, v. 75, n. 2, p. 87–106, 1987. ISSN 10902651. Available: <https://doi.org/10.1016/0890-5401(87)90052-6>. Accessed: 05/01/2021. Citations on pages 45 and 46.

BIERMANN, A. W.; FELDMAN, J. A. On the Synthesis of Finite-State Machines from Samples of Their Behavior. **IEEE Transactions on Computers**, C-21, n. 6, p. 592–597, 1972. ISSN 00189340. Available: <https://doi.org/10.1109/TC.1972.5009015>. Accessed: 09/01/2021. Citation on page 48.

BOLLIG, B.; HABERMEHL, P.; KERN, C.; LEUCKER, M. Angluin-style learning of NFA. In: **IJCAI International Joint Conference on Artificial Intelligence**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009. (IJCAI'09), p. 1004–1009. ISBN 9781577354260. ISSN 10450823. Citations on pages 23, 24, 30, 45, 46, and 47.

BOURQUE, P.; FAIRLEY, R. E. **Guide to the Software Engineering - Body of Knowledge (SWEBOK 2014).** Los Alamitos, CA: IEEE Computer Society, 2014. 346 p. ISSN 07407459. ISBN 0-7695-2330-7. Available: <www.swebok.org>. Accessed: 05/02/2021. Citation on page 33.

BüCHLER, M.; HOSSEN, K.; MIHANCEA, P. F.; MINEA, M.; GROZ, R.; ORIAT, C. Model inference and security testing in the spacios project. In: **2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)**. [s.n.], 2014. p. 411–414. Available: <https://10.1109/CSMR-WCRE.2014.6747207>. Citation on page 67.

COOK, J. E.; WOLF, A. L. Discovering models of software processes from event-based data. **ACM Transactions on Software Engineering and Methodology**, Association for Computing Machinery, New York, NY, USA, v. 7, n. 3, p. 215–249, 1998. ISSN 1049331X. Available: <https://doi.org/10.1145/287000.287001>. Accessed: 11/01/2021. Citations on pages 24, 30, 31, 32, 47, and 48.

DAMASCENO, C. D. N.; MASIERO, P. C.; SIMAO, A. Evaluating test characteristics and effectiveness of FSM-based testing methods on RBAC systems. In: **ACM International Conference Proceeding Series**. New York, NY, USA: Association for Computing Machinery, 2016. (SBES '16), p. 83–92. ISBN 9781450342018. Available: <https://doi.org/10.1145/2973839.2973849>. Accessed: 02/01/2021. Citation on page 40.

DAS, S.; MOZER, M. A Unified Gradient-Descent/Clustering Algorithm Architecture for Finite State Machine Induction. In: **Advances in Neural Information Processing Systems 6**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

(NIPS'93), p. 19–26. Available: <https://dl.acm.org/doi/10.5555/2987189.2987192>. Accessed: 12/01/2021. Citation on page 48.

DAVIS, A. M. A comparison of techniques for the specification of external system behavior. **Communications of the ACM**, Association for Computing Machinery, New York, NY, USA, v. 31, n. 9, p. 1098–1115, 1988. ISSN 15577317. Available: <https://doi.org/10.1145/48529.48534>. Accessed: 01/01/2021. Citations on pages 24, 28, 29, 37, and 38.

EL-FAKIH, K.; GROZ, R.; IRFAN, M. N.; SHAHBAZ, M. Learning Finite State Models of Observable Nondeterministic Systems in a Testing Context. In: **22nd IFIP International Conference on Testing Software and Systems**. [s.n.], 2010. p. 97–102. Available: <http://publica.fraunhofer.de/documents/N-148900.html>. Accessed: 07/01/2021. Citation on page 46.

FUJIWARA, S.; BOCHMANN, G. v.; KHENDEK, F.; AMALOU, M.; GHEDAMSI, A. Test selection based on finite state models. **IEEE Transactions on Software Engineering**, IEEE Press, v. 17, n. 6, p. 591–603, 1991. ISSN 00985589. Available: <https://doi.org/10.1109/32.87284>. Accessed: 03/02/2021. Citations on pages 24, 29, 38, and 39.

GILL, A. **Introduction to the theory of finite-state machines**. New York: McGraw-Hill, 1962. ISBN 0070232431. Citation on page 38.

GINSBURG, S. Synthesis of Minimal-State Machines. **IRE Transactions on Electronic Computers**, EC-8, n. 4, p. 441–449, 1959. ISSN 03679950. Available: <https://doi.org/10.1109/TEC.1959.5222056>. Accessed: 10/01/2021. Citation on page 47.

GOLD, E. M. Language identification in the limit. **Information and Control**, v. 10, n. 5, p. 447–474, 1967. ISSN 00199958. Available: <https://doi.org/10.1016/S0019-9958(67)91165-5>. Accessed: 15/02/2021. Citation on page 31.

______. System identification via state characterization. **Automatica**, v. 8, n. 5, p. 621–636, 1972. ISSN 00051098. Available: <https://doi.org/10.1016/0005-1098(72)90033-7>. Accessed: 06/01/2021. Citation on page 45.

______. Complexity of automaton identification from given data. **Information and Control**, v. 37, n. 3, p. 302–320, 1978. ISSN 00199958. Available: <https://doi.org/10.1016/S0019-9958(78)90562-4>. Accessed: 09/01/2021. Citation on page 47.

GROZ, R.; BREMOND, N.; SIMAO, A. Using adaptive sequences for learning non-resettable fsms. In: UNOLD, O.; DYRKA, W.; WIECZOREK, W. (Ed.). **Proceedings of The 14th International Conference on Grammatical Inference 2018**. PMLR, 2019. (Proceedings of Machine Learning Research, v. 93), p. 30–43. Available: <https://proceedings.mlr.press/v93/groz19a.html>. Citation on page 52.

GROZ, R.; BREMOND, N.; SIMAO, A.; ORIAT, C. hW-inference: A heuristic approach to retrieve models through black box testing. **Journal of Systems and Software**, Elsevier {BV}, v. 159, p. 110426, 2020. ISSN 01641212. Available: <https://doi.org/10.1016/j.jss.2019.110426>. Accessed: 31/12/2020. Citations on pages 24, 30, 39, 45, 46, 51, 52, 53, and 68.

GROZ, R.; SIMAO, A.; BREMOND, N.; ORIAT, C. Revisiting ai and testing methods to infer fsm models of black-box systems. In: **Proceedings of the 13th International Workshop on Automation of Software Test**. New York, NY, USA: Association for Computing Machinery, 2018. (AST '18), p. 16–19. ISBN 9781450357432. Available: <https://doi.org/10.1145/3194733.3194736>. Citation on page 52.

GROZ, R.; SIMAO, A.; PETRENKO, A.; ORIAT, C. Inferring finite state machines without reset using state identification sequences. In: EL-FAKIH, K.; BARLAS, G.; YEVTUSHENKO, N. (Ed.). **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. Cham: Springer International Publishing, 2015. v. 9447, p. 161–177. ISBN 9783319259444. ISSN 16113349. Available: <https://doi.org/10.1007/978-3-319-25945-1_10>. Accessed: 06/01/2021. Citation on page 46.

HALM, M. **The hW-Inference Algorithm: Theory and Application**. Master's Thesis (Master's Thesis) — Fakultät für Informatik, 2021. Available: <https://sdq.kastel.kit.edu/institutsseminar/The_hW-inference_Algorithm:_Theory_and_Application>. Citations on pages 25, 44, 54, 55, and 56.

HALM, M.; BRAZ, R. S.; GROZ, R.; ORIAT, C.; SIMAO, A. Improving model inference via w-set reduction. In: CLARK, D.; MENENDEZ, H.; CAVALLI, A. R. (Ed.). **Testing Software and Systems**. Cham: Springer International Publishing, 2022. p. 90–105. ISBN 978-3-031-04673-5. Available: <https://doi.org/10.1007/978-3-031-04673-5_7>. Citations on pages 52, 53, 68, and 71.

HEERDT, G. van. **An Abstract Automata Learning Framework**. Master's Thesis (Master's Thesis) — Master's thesis, Radboud Universiteit Nijmegen, 2016. Available: <http://www.calf-project.org/publications/Gerco-thesis.pdf>. Accessed: 09/01/2021. Citations on pages 24, 30, and 47.

HIERONS, R. M. Generating candidates when testing a deterministic implementation against a non-deterministic finite-state machine. **Computer Journal**, Oxford University Press ({OUP}), v. 46, n. 3, p. 307–318, 2003. ISSN 00104620. Available: <https://doi.org/10.1093/comjnl/46.3.307>. Accessed: 27/12/2020. Citations on pages 41, 42, and 43.

______. Testing from a nondeterministic finite state machine using adaptive state counting. **IEEE Transactions on Computers**, v. 53, n. 10, p. 1330–1342, 2004. ISSN 00189340. Available: <https://doi.org/10.1109/TC.2004.85>. Accessed: 27/12/2020. Citation on page 40.

IPATE, F. Bounded sequence testing from non-deterministic finite state machines. In: **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. Springer Berlin Heidelberg, 2006. v. 3964 LNCS, p. 55–70. ISBN 9783540341840. Available: <https://doi.org/10.1007/11754008_4>. Accessed: 01/01/2021. Citation on page 43.

ISO/IEC/IEEE Approved Draft International Standard - Systems and Software Engineering - Vocabulary. **Iso/Iec/Ieee P24765/D3:2017**, p. 1–570, 2017. Available: <https://doi.org/10.1109/IEEESTD.2017.8016712>. Accessed: 07/02/2021. Citations on pages 27 and 33.

KHALILI, A.; TACCHELLA, A. Learning Nondeterministic Mealy Machines. In: CLARK, A.; KANAZAWA, M.; YOSHINAKA, R. (Ed.). **Proceedings of the 12th International Conference on Grammatical Inference**. Kyoto, Japan: PMLR, 2014. (Proceedings of Machine Learning Research, v. 34), p. 109–123. Available: <http://proceedings.mlr.press/v34/khalili14a.html>. Accessed: 05/01/2021. Citations on pages 40, 41, and 46.

LANG, K. J. Random DFA's can be approximately learned from sparse uniform examples. In: **Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory**. New York, NY, USA: Association for Computing Machinery, 1992. (COLT '92), p. 45–52. ISBN 089791497X. Available: <https://doi.org/10.1145/130385.130390>. Accessed: 11/01/2021. Citation on page 47.

LEE, D.; YANNAKAKIS, M. Principles and methods of testing finite state machines-a survey. **Proceedings of the IEEE**, v. 84, n. 8, p. 1090–1123, Aug 1996. ISSN 1558-2256. Available: <https://doi.org/10.1109/5.533956>. Citation on page 52.

LORENZOLI, D.; MARIANI, L.; PEZZÈ, M. Automatic generation of software behavioral models. In: **Proceedings - International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2008. (ICSE '08), p. 501–510. ISBN 9781605580791. ISSN 02705257. Available: <https://doi.org/10.1145/1368088.1368157>. Accessed: 28/01/2021. Citation on page 48.

LUCAS, S. M.; REYNOLDS, T. J. Learning DFA: Evolution versus evidence driven state merging. In: **2003 Congress on Evolutionary Computation, CEC 2003 - Proceedings**. [s.n.], 2003. v. 1, p. 351–358. Available: <https://doi.org/10.1109/CEC.2003.1299597>. Accessed: 10/01/2021. Citation on page 47.

MEINKE, K.; WALKINSHAW, N. Model-based testing and model inference. In: MARGARIA, T.; STEFFEN, B. (Ed.). **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. v. 7609 LNCS, n. PART 1, p. 440–443. ISBN 9783642340253. ISSN 03029743. Available: <https://doi.org/10.1007/978-3-642-34026-0_32>. Accessed: 14/02/2021. Citations on pages 23 and 30.

MOORE, E. F. Gedanken-Experiments on Sequential Machines. In: SHANNON, C. E.; MCCARTHY, J. (Ed.). **Automata Studies. (AM-34)**. Princeton University Press, 1956. p. 129–154. Available: <https://doi.org/10.1515/9781400882618-006>. Accessed: 14/02/2021. Citation on page 33.

Naeem Irfan, M.; ORIAT, C.; GROZ, R. Model Inference and Testing. In: MEMON, A. B. T. A. i. C. (Ed.). **Advances in Computers**. Elsevier, 2013. v. 89, p. 89–139. ISBN 0065-2458. Available: <https://doi.org/10.1016/B978-0-12-408094-2.00003-5>. Accessed: 11/02/2021. Citations on pages 23, 24, 29, 30, and 33.

NEIDER, D.; SMETSERS, R.; VAANDRAGER, F.; KUPPENS, H. Benchmarks for automata learning and conformance testing. In: _____. **Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday**. Cham: Springer International Publishing, 2019. p. 390–416. ISBN 978-3-030-22348-9. Available: <https://doi.org/10.1007/978-3-030-22348-9_23>. Citation on page 76.

NOVELLA, L.; TUFO, M.; FIENGO, G. Improving test suites via a novel testing with model learning approach. In: **Proceedings - 2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2018**. [s.n.], 2018. p. 235–240. ISBN 9781538669167. ISSN 1524-4547. Available: <https://doi.org/10.1109/WETICE.2018.00051>. Accessed: 11/02/2021. Citations on pages 23 and 29.

ONCINA, J.; GARCÍA, P. Inferring Regular Languages in Polynomial Updated Time. In: _____. **Pattern Recognition and Image Analysis**. [s.n.], 1992. p. 49–61. Available: <https://doi.org/10.1142/9789812797902_0004>. Accessed: 11/01/2021. Citation on page 47.

PACHAROEN, W.; AOKI, T.; BHATTARAKOSOL, P.; SURARERKS, A. Active learning of nondeterministic finite state machines. **Mathematical Problems in Engineering**, Hindawi Publishing Corporation, v. 2013, p. 373265, 2013. ISSN 1024123X. Available: <https://doi.org/10.1155/2013/373265>. Accessed: 05/01/2021. Citations on pages 24, 30, 40, 45, and 46.

PETRENKO, A.; AVELLANEDA, F. Learning and Adaptive Testing of Nondeterministic State Machines. In: **Proceedings - 19th IEEE International Conference on Software Quality, Reliability and Security, QRS 2019**. [s.n.], 2019. p. 362–373. ISBN 9781728139272. Available: <https://doi.org/10.1109/QRS.2019.00053>. Accessed: 27/12/2020. Citation on page 46.

PETRENKO, A.; AVELLANEDA, F.; GROZ, R.; ORIAT, C. From passive to active FSM inference via checking sequence construction. In: YEVTUSHENKO, N.; CAVALLI, A. R.; YENIGÜN, H. (Ed.). **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. Cham: Springer International Publishing, 2017. v. 10533 LNCS, p. 126–141. ISBN 9783319675480. ISSN 16113349. Available: <https://doi.org/10.1007/978-3-319-67549-7_8>. Accessed: 10/01/2021. Citation on page 47.

PETRENKO, A.; YEVTUSHENKO, N.; LEBEDEV, A.; DAS, A. Nondeterministic state machines in protocol conformance testing. In: RAFIQ, O. (Ed.). **IFIP Transactions C: Communication Systems**. [S.l.]: North-Holland, 1994. ({IFIP} Transactions, C-19), p. 363–378. ISBN 0444816976. ISSN 0926549X. Citations on pages 32 and 42.

PHILAE, A. **Data gathering, preparation and format. Tech. rep. WP1/D1.1**. [S.l.], 2019. Available: <https://projects.femto-st.fr/philae/>. Citation on page 68.

PINHEIRO, P. V. P. **Teste baseado em modelos para serviços RESTful usando máquinas de estados de protocolos UML**. Master's Thesis (Master's Thesis) — Universidade de São Paulo (USP), 2014. Available: <https://doi.org/10.11606/d.55.2014.tde-14072014-165410>. Accessed: 04/02/2021. Citation on page 34.

PRADEL, M.; GROSS, T. R. Automatic generation of object usage specifications from large method traces. In: **ASE2009 - 24th IEEE/ACM International Conference on Automated Software Engineering**. [s.n.], 2009. p. 371–382. ISBN 9780769538914. ISSN 1938-4300. Available: <https://doi.org/10.1109/ASE.2009.60>. Accessed: 30/01/2021. Citation on page 48.

PRESSMAN, R. S. **Engenharia de Software - Uma Abordagem Profissional**. 7. ed. Porto Alegre: Grupo A - AMGH, 2011. 779 p. ISSN 85860457. ISBN 978-85-8055-044-3. Citations on pages 32, 33, and 34.

RIVEST, R. L.; SCHAPIRE, R. E. Inference of finite automata using homing sequences. In: _____. **Machine Learning: From Theory to Applications: Cooperative Research at Siemens and MIT**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993. p. 51–73. ISBN 978-3-540-47568-2. Available: <https://doi.org/10.1007/3-540-56483-7_22>. Accessed: 06/01/2021. Citation on page 46.

SHAHBAZ, M.; GROZ, R. Inferring mealy machines. In: **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. Springer Berlin Heidelberg, 2009. v. 5850 LNCS, p. 207–222. ISBN 3642050883. Available: <https://doi.org/10.1007/978-3-642-05089-3_14>. Accessed: 06/01/2021. Citations on pages 45 and 46.

SIMÃO, A.; PETRENKO, A.; MALDONADO, J. C. Comparing finite state machine test coverage criteria. **IET Software**, Institution of Engineering and Technology ({IET}), v. 3, n. 2, p. 91–105, 2009. ISSN 17518806. Available: <https://doi.org/10.1049/iet-sen.2008.0018>. Accessed: 02/01/2021. Citations on pages 37, 39, and 44.

SPITSYNA, N.; EL-FAKIH, K.; YEVTUSHENKO, N. Studying the separability relation between finite state machines. **Software Testing Verification and Reliability**, v. 17, n. 4, p. 227–241, 2007. ISSN 09600833. Available: <https://doi.org/10.1002/stvr.374>. Accessed: 05/01/2021. Citation on page 41.

TSAREV, F.; EGOROV, K. Finite state machine induction using genetic algorithm based on testing and model checking. In: **Genetic and Evolutionary Computation Conference, GECCO'11 - Companion Publication**. New York, NY, USA: Association for Computing Machinery, 2011. (GECCO '11), p. 759–762. ISBN 9781450306904. Available: <https://doi.org/10.1145/2001858.2002085>. Accessed: 10/01/2021. Citation on page 47.

UTTING, M.; LEGEARD, B. **Practical model-based testing: a tools approach**. San Francisco, CA: Morgan Kaufmann Publishers, 2007. 433 p. ISBN 9780123725011. Citation on page 32.

UTTING, M.; LEGEARD, B.; DADEAU, F.; TAMAGNAN, F.; BOUQUET, F. Identifying and generating missing tests using machine learning on execution traces. In: **2020 IEEE International Conference On Artificial Intelligence Testing (AITest)**. [s.n.], 2020. p. 83–90. Available: <https://doi.org/10.1109/AITEST49225.2020.00020>. Citation on page 68.

UTTING, M.; PRETSCHNER, A.; LEGEARD, B. A taxonomy of model-based testing approaches. **Software Testing Verification and Reliability**, v. 22, n. 5, p. 297–312, 2012. ISSN 09600833. Available: <https://doi.org/10.1002/stvr.456>. Accessed: 24/02/2021. Citations on pages 33 and 35.

WAGNER, F.; SCHMUKI, R.; WAGNER, T.; WOLSTENHOLME, P. **Modeling software with finite state machines: A practical approach**. Boca Raton, FL: Auerbach, 2006. 1–370 p. ISBN 9781420013641. Available: <https://doi.org/10.1201/9781420013641>. Accessed: 01/01/2021. Citations on pages 29 and 37.

WALKINSHAW, N.; BOGDANOV, K.; ALI, S.; HOLCOMBE, M. Automated discovery of state transitions and their functions in source code. **Software Testing Verification and Reliability**, v. 18, n. 2, p. 99–121, 2008. ISSN 09600833. Available: <https://doi.org/10.1002/stvr.380>. Accessed: 14/02/2021. Citations on pages 23 and 30.

WALKINSHAW, N.; LAMBEAU, B.; DAMAS, C.; BOGDANOV, K.; DUPONT, P. STAMINA: A competition to encourage the development and assessment of software model inference techniques. **Empirical Software Engineering**, v. 18, n. 4, p. 791–824, 2013. ISSN 13823256. Available: <https://doi.org/10.1007/s10664-012-9210-3>. Accessed: 12/01/2021. Citations on pages 23, 27, 28, and 47.

WALKINSHAW, N.; TAYLOR, R.; DERRICK, J. Inferring extended finite state machine models from software executions. **Empirical Software Engineering**, v. 21, n. 3, p. 811–853, 2016. ISSN 15737616. Available: <https://doi.org/10.1007/s10664-015-9367-7>. Accessed: 28/01/2021. Citation on page 48.

WEYUKER, E. J. Assessing Test Data Adequacy through Program Inference. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, Association for Computing Machinery, New York, NY, USA, v. 5, n. 4, p. 641–655, 1983. ISSN 15584593. Available: <https://doi.org/10.1145/69575.357231>. Accessed: 15/02/2021. Citation on page 33.

ZHIRABOK, A. N.; KALININA, N. A.; SHUMSKII, A. E. Method for the Functional Diagnosis of Nondeterministic Finite State Machines. **Journal of Computer and Systems Sciences International**, Pleiades Publishing Ltd, v. 59, n. 4, p. 565–574, 2020. ISSN 15556530. Available: <https://doi.org/10.1134/s1064230720040152>. Accessed: 05/01/2021. Citation on page 38.

ICMC USP
SÃO CARLOS