Advances in Model Learning for Software Systems

Rick Smetsers

Advances in Model Learning for Software Systems

Proefschrift

ter verkrijging van de graad van doctor

aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.H.J.M. van Krieken,

volgens besluit van het college van decanen
in het openbaar te verdedigen op donderdag 29 maart 2018
om 10:30 uur precies

door

Rick Henricus Adrianus Maria Smetsers

geboren op 6 november 1989
te Eindhoven

Het onderzoek in dit proefschrift is uitgevoerd onder auspiciën van SIKS, de Nederlandse Onderzoekschool voor Informatie- en Kennissystemen.

Dit proefschrift is onderdeel van het onderzoeksprogramma *Learning Extended State Machines for Malware Analysis* (LEMMA), dat is gefinancierd door de Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO).

ISBN: 978-94-92896-11-7
SIKS Dissertation Series № 2018-08

Advances in Model Learning for Software Systems

Doctoral Thesis

to obtain the degree of doctor

from Radboud University Nijmegen
on the authority of the Rector Magnificus prof. dr. J.H.J.M. van Krieken,

according to the decision of the Council of Deans
to be defended in public on Thursday, March 29, 2018
at 10:30 hours

by

Rick Henricus Adrianus Maria Smetsers

born on November 6, 1989
in Eindhoven

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

This work is part of the research programme on Learning Extended State Machines for Malware Analysis (LEMMA), which is financed by the Netherlands Organisation for Scientific Research (NWO).

ISBN: 978-94-92896-11-7
SIKS Dissertation Series № 2018-08

Samenvatting

Softwaresystemen worden steeds complexer. Hierdoor neemt het aantal fouten toe. Het *leren van modellen* wordt meer en meer een toegankelijke techniek om deze fouten op te sporen. Algoritmen voor het leren van modellen kijken niet naar de interne structuur, maar naar het *gedrag* van een systeem. Het ontwerpen van algoritmen hiervoor is belangrijk om de techniek beter toegankelijk en toepasbaar te maken. Dit proefschrift presenteert hiervoor de volgende bijdragen:

1. Algoritmen voor het leren van modellen maken iteratief een *hypothese* voor het systeem in kwestie. Wij zijn de eersten die het begrip *kwaliteit* formaliseren voor deze hypotheses, en we presenteren een leeralgoritme dat er voor zorgt dat de kwaliteit van een hypothese niet afneemt (Hoofdstukken 3 en 4).

2. Hypothesen worden verfijnd door *tegenvoorbeelden*. Inputsequenties die de *toestanden* van een hypothese onderscheiden spelen een belangrijke rol in het vinden van deze tegenvoorbeelden. Wij presenteren een algoritme dat deze onderscheidende sequenties efficient construeert (Hoofdstuk 2). Daarnaast introduceren we *fuzzing* als een nieuwe, aanvullende techniek voor het vinden van tegenvoorbeelden (Hoofdstuk 6).

3. We leggen uit hoe verschillende modelformalismen en observaties van hun gedrag uitgedrukt kunnen worden in een *logische formule*, en we laten zien dat bestaande methoden voor *satisfiabiilty modulo theories* gebruikt kunnen worden om hiervan een hypothese te maken (Hoofdstuk 5). Deze nieuwe aanpak voor het leren van modellen gebruikt geen overtollige inputsequenties, die wel aanwezig zijn als de klassieke aanpak toegepast wordt in de praktijk.

4. Een eerste vereiste voor het leren van modellen is dat het inputformaat bekend is. Wij geven een overzicht van de programma's en technieken die gebruikt kunnen worden om het inputformaat te *reverse engineeren*, en de toepassingen hiervan voor *digitale veiligheid*.

Summary

Software systems are becoming increasingly complex. This leads to an inevitable increase in *bugs*. *Model learning* is becoming a popular technique for finding these bugs. Instead of viewing a software system via its internal structure, model learning algorithms construct a model from observations of the system's *behaviour*. The design of algorithms for models and model learning is a fundamental problem, and research in the area is important for making the techniques more applicable and accessible. This thesis presents the following contributions in this regard:

1. Model learning algorithms work by iteratively constructing a *hypothesis* for the system under learning. We are the first to formalize the notion of *quality* for these hypotheses, and we present a modification for the learning process that ensures that the quality of subsequent hypotheses never decreases (Chapters 3 and 4).

2. Hypotheses are refined by finding *counterexamples* for them. Input sequences that separate the *states* of an hypothesis play an important role in finding counterexamples. We present an algorithm for constructing these separating sequences efficiently (Chapter 2). In addition, we introduce *fuzzing* as a new, complementary technique for finding counterexamples (Chapter 6).

3. We explain how different model formalisms and observations of their behaviour can be expressed in a *logic formula*, and we show that *satisfiability modulo theories* solvers can be used to construct a hypothesis from this formula (Chapter 5). This new approach to model learning removes redundancy that is present in the classical approach when applied in practice.

4. A prerequisite for model learning is that the system's *input format* is known. We give an overview of the tools and techniques for *reverse engineering* the input format, and their applications in *security* (Chapter 7).

Contents

Chapter 1

Introduction

Software systems are becoming increasingly complex, and with this increase in complexity comes an inevitable increase of *bugs*.

The Linux kernel, for example, was growing faster than ever in 2016, gaining on average nearly 11 files and 4 600 lines of code every day [132]. Launched in 1991, the first Linux kernel had little over 10 thousand lines of code. Today, it has over 22 million. Currently, there are more than 5 thousand known bugs in the Linux kernel, and possibly even more that are not detected yet. The number of new detected bugs is increasing too, with more than double the number of bugs reported in 2016, compared to 2015 [133].

In 2012, a similar number of lines of code (24 million) could be found in an F-35, the combat aircraft developed in the Joint Strike Fighter (JSF) program [13]. Already then, this was 9 million lines more than originally envisioned. Christopher Bogdan, the JSF program head, warned that *software* is the riskiest facet of F-35 development and the most likely cause of delays [121]. Being the most expensive military weapons system in history, these delays have far-reaching consequences. By 2014, the program was "163 billion dollars over budget [and] seven years behind schedule" [50].

A possible explanation for the increase in bugs and unforeseen delays is that traditional *formal methods* for the *verification* of a system can not cope with the aforementioned increase in complexity. Formal methods are a particular kind of mathematically based techniques that contribute to the reliability and robustness of a system's design. The goal of formal methods is to prove properties of the *specification* of a system, or to show that *requirements* of the system hold. Often, formal methods require a *model* of the system to be scalable and effective. Unfortunately, the time and effort required to develop a model grows disproportionately as the system increases in size. As such, the construction of models is often omitted during software development due to the cost involved in generating and maintaining them [129].

An alternative to constructing models manually is to *learn* (i.e. reverse engineer) them. One way to approach this is by viewing a system not via its internal structure, but through the laws which govern its *behaviour*. A type of model that is well suited for describing the behaviour of software and hardware systems is an *automaton*. An automaton is a mathematical model of computation that can represent the *states* of a system and *transitions* between these states.

The topic of learning automata has long been researched from a theoretical perspective in a subfield of computer science that is known to as *grammatical inference* (dating back to the seminal works by Moore [104], Gold [63] and Angluin [11]). Only in recent years, researchers have recognized the practical avail of this work [114, 117, 71, 131]. This has led to the inception of *model learning* [140]. The field of model learning is concerned with the design and application of algorithms that automatically construct models of (software) systems from their input-output behaviour. Model learning is a fundamental problem in software science, and research in the area is important for driving the field forward. This thesis presents several solutions for open problems in the field that make model learning more practically applicable for reverse engineering software systems.

The remainder of this introduction is structured as follows. First, we give a gentle introduction model learning for software systems in Section 1.1. Then, we outline the contributions presented in this thesis in Section 1.2. We conclude this introduction with an overview of related work in Section 1.3.

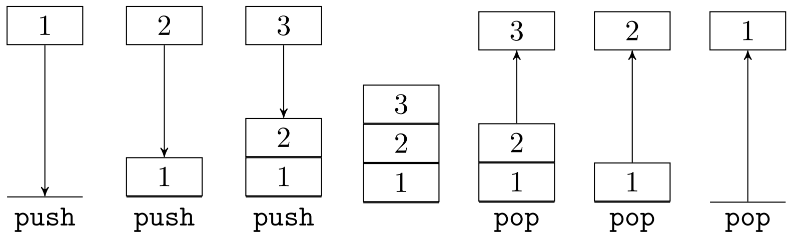


Figure 1.1: A stack with *push* and *pop* operations. The operation that is executed is shown at the bottom, and the element is shown at the top. The contents of the stack are shown in the middle.

1.1 Model Learning for Software Systems

It seems that we humans have an innate ability to learn the behavior of software and hardware systems by simply interacting with them and observing the resulting behavior. Most of us could learn how to operate a new TV without ever consulting the manual, for example. The field of *model learning* is concerned with the design and application of algorithms that automatically construct a model from observations of a system's behaviour.

In this section, we explain the core concepts for model learning, and we introduce the key algorithms. Where possible, we will illustrate these concepts by means of the following running example.

Example 1.1. *A* stack *is an abstract data type for storing data according to the* last-in first-out *(LIFO) principle. It has two operations:* `push`(x)*, which adds an element x to the stack, and* `pop`*, which removes the most recently added element that was not yet removed. Figure 1.1 illustrates these operations on a stack.*

1.1.1 Systems, Interfaces and Protocols

A *system* is defined by the Merriam-Webster dictionary as "a regularly interacting or interdependent group of items forming a unified whole" [101]. We think of a system in the context of *hardware* and *software*, where a system consists of several separate hardware components, computer programs and associated configuration files that operate together to accomplish a certain task [129, Chapter 1.1.1].

Example 1.2. *A stack can be considered as a part of a system. It consists of an underlying data structure and a set of methods that perform its operations. Figure 1.2 shows an implementation of a stack.*

The way that a system can be interacted with is defined by its *interface*. The interface describes the *inputs* that are understood by the system (e.g. methods, messages) and the type of *data parameters* that these inputs may be supplied with.

Example 1.3. *The interface for a stack consists of the method invocations* `push` *and* `pop`*. The* `push` *input takes a single data parameter of type* **elem** *and does not return an output (i.e. it returns* **void***). The* `pop` *input does not take any parameters but instead returns a data value of type* **elem** *as an output.*

The user of a system does not have to understand how a system is implemented. Instead, he has to understand the rules for how the system can be interacted with. These rules are described by the *protocol* of a system.

Example 1.4. *The protocol of a stack can be described as follows:*

> *If the stack is not full, the* `push` *operation can be called to add an element to the top of the stack. If the stack is not empty, the* `pop` *operation returns the top element of the stack.*

A textual description of a protocol can ambiguous or incomplete. What happens if the `push` *operation is called on a full stack? And what happens if the* `pop` *operation is called on an empty stack?*

1.1.2 Models

The protocol of a system can be described in a concise, complete and unambiguous way by a *model*. A model is an abstract representation of a protocol that can help people understand it better, and that can be used to verify some of its properties, for example. Often, however, the construction of such models is omitted during software development due to the time and cost involved in generating and maintaining them [129]. This is where model learning comes in.

```
type elem

class stack
    data : array of type elem
    top : int
    capacity : int

    constructor(capacity : int)
        this.data ← empty array of size capacity
        this.top ← 0
        this.capacity ← capacity

    void push(element : elem)
        if this.top = this.capacity then
            raise overflow error
        else
            this.data[this.top] ← element
            this.top ← this.top + 1

    elem pop()
        if this.top = 0 then
            raise underflow error
        else
            this.top ← this.top − 1
            element ← this.data[this.top]
            return element
```

Figure 1.2: Implementation of a stack

The goal of model learning is to automatically construct a model of the protocol of a system. Specifically, it aims to discover (some of) the control-specific and data-dependence relationships that are present in the system by interacting with it and/ or observing its behaviour.

Different flavours of *automata* are extremely well suited to describe these relationships. Conceptually, an automaton is a transition system that consists of a finite set of *states*, and a set of labeled *transitions* between these states. States model system configurations, and transitions model how states change over time. The automaton starts in an *initial* state and is always in one state at a time, its *current* state. Transitions between states are triggered by *inputs*. In addition, transitions and/ or states can produce *outputs*. As such, an automaton can be used to describe the outputs that a system produces in response to *sequences* of inputs.

Different types of automata exist, that can be classified according to the following dimensions.

Determinism. An automaton is *deterministic* if (a) each transition is uniquely determined by its source state and input, and (b) an input is required for triggering a transition. An automaton is *nondeterminsitic* if one of these restrictions does not apply.

Probability. An automaton is *probabilistic* if the outcome of a transition can be described by a probability distribution. Typically, this distribution is determined by the current state of the automaton. Automata that are not probabilistic are called *non-probabilistic*.

Output. An automaton is an *acceptor* if it produces a single binary output for a sequence of inputs, indicating whether or not the sequence describes proper behaviour of the system. A probabilistic automaton describes the probability that a sequence of inputs occurs. This is typically the product of probabilities on the transitions that were triggered along the way. An automaton is a *transducer* if it produces outputs based on each input, or current state.

Memory. Some flavours of automata have a finite amount of *registers* in which they can store data values attached to inputs for later comparison. Others have clocks that allow to test the lapse of time between two events. Both of these flavours are called *extended finite* *automata.* In an extended finite automaton, a transition can be expressed by an *if* statement consisting of a set of trigger conditions. If the trigger conditions are all satisfied, the transition is fired, bringing the automaton from the current state to the next state and performing the specified data operations (e.g. storing a value or setting a clock).

In this thesis, we are mainly concerned with deterministic and non-probabilistic automata. We distinguish between three types of problems that can be modelled by (different flavours of) deterministic, non-probabilistic automata:

1. Distinguish between valid and invalid sequences of inputs.
2. Describe control-specific input/output behaviour.
3. Characterize data-dependent relationships between inputs and outputs.

Despite their differences, model learning algorithms for these problems are remarkably similar. Let us introduce the most well known types of automata for each problem, the *deterministic finite automaton* (DFA), *Mealy machine* and *register automaton* (RA). Most model learning algorithms, as well as the contributions in this thesis, focus on these formalisms.

Deterministic finite automata

A DFA is an automaton that accepts (and rejects) valid (and invalid) sequences of inputs. Its states are either accepting or rejecting, and the transitions between states are labeled with inputs. A state can not have two outgoing transitions with the same input. A DFA has a single distinguished initial state in which all sequences start. Upon receiving an input, it transitions to the corresponding next state. If the automaton is in an accepting state when no inputs are left, then the sequence of inputs was valid. If it ends in a rejecting state, the sequence was invalid. Therefore, a DFA is a deterministic, non-probabilistic acceptor.

The semantics of a DFA can be described as a set of input sequences. Therefore, they are well suited for solving the first problem of distinguishing between valid and invalid sequences of inputs.

Formally, a DFA can be described by a tuple (I, Q, q_0, δ, F), where:

- I is a finite set of inputs,
- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- $\delta : Q \times I \to Q$ is a transition function from states and inputs to states, and
- $F \subseteq Q$ is the set of *accepting states*.

A DFA is deterministic and *input complete* by definition (of the transition function δ). An automaton is *input complete* if it defines a transition for each input in each state.

A *computation* of a DFA A on a sequence of inputs $x = x_1 \dots x_{|x|}$ can be described as a sequence of states $q'_0 \dots q'_{|x|}$, where

1. $q'_0 = q_0$, and
2. $q'_i = \delta(q'_{i-1}, x_i)$ for $1 \leq i \leq |x|$

DFA A *accepts* x if its computation ends in an accepting state, i.e. $q'_{|x|} \in F$. It is said to *reject* x otherwise.

Let S_+ be a set of sequences that should be accepted, and let S_- be a disjoint set of sequences that should be rejected. Let S be the set that contains all of these sequences, along with their labels, i.e. $S = \{(x, \mathit{true}) : x \in S_+\} \cup \{(x, \mathit{false}) : x \in S_-\}$. A DFA is *consistent with* S if it accepts all sequences in S_+, and rejects all sequences in S_-.

Example 1.5. *A DFA can be used to describe whether sequences of method calls on a stack are error-free or not. Data parameters and outputs are of no importance for this purpose. Figure 1.3 shows such a DFA for a stack of size 2. Here, vertices represent states and labeled edges represent transitions. The initial state is represented by the unlabeled edge, and accepting states are represented by double circles. This DFA accepts all sequences of method calls that do not yield an error, i.e. in which* `pop` *is not called on an empty stack, and* `push` *is not called on a full stack.*

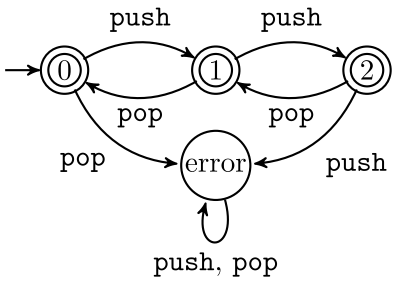


Figure 1.3: A DFA for a stack of size 2

Mealy machines

A Mealy machine is a transducer that can be used to describe the input/output behaviour of systems. Upon receiving an input, it produces an output and transitions to the next state. As such, a Mealy machine can be used to describe *traces*, which are sequences of alternating inputs and outputs.

Unlike a DFA, the semantics of a Mealy machine can not be described by a set of input sequences. Instead, it can be seen as a mapping from input sequences to output sequences. Therefore, Mealy machines are suited for modeling control-specific input/output behaviour of a system.

Formally, a Mealy machine can be described by a tuple $(I, O, Q, q_0, \delta, \lambda)$, where:

– I is a finite set of inputs,

– O is a finite set of outputs,

– Q is a finite set of states,

– $q_0 \in Q$ is the initial state,

– $\delta : Q \times I \to Q$ is a transition function that maps states and inputs to states, and

– $\lambda : Q \times I \to O$ is an output function that maps states and inputs to outputs.

Like a DFA, a Mealy machine is deterministic and input complete by definition.

A *trace* can be described by a pair (x, y), where $x \in I^*$ is an input sequence and $y \in O^*$ is an output sequence of equal length. A Mealy machine M *generates* y when provided with x if there exists a sequence of states $q'_0 \dots q'_{|x|}$ such that:

1. $q'_0 = q_0$,
2. $q'_i = \delta(q'_{i-1}, x_i)$ for $1 \leq i \leq |x|$, and
3. $\lambda(q'_{i-1}, x_i) = y_i$ for $1 \leq i \leq |x|$.

Let S be a set of traces, then a Mealy machine is consistent with S if for each (x, y) in S it generates y when provided with x.

Example 1.6. *A Mealy machine can be used to describe input/output behaviour of a stack if we restrict the possible data parameters and values to a finite set. Figure 1.4 shows such a Mealy machine for a stack of size 2 that can contain elements* `0` *and* `1`*. Where applicable, the inputs for this Mealy machine are taken to be the product of the method calls and these elements, i.e.* $I = \{\texttt{push 0}, \texttt{push 1}, \texttt{pop}\}$*. The outputs are taken to be the actual responses to the method calls, i.e.* $O = \{\mathit{error}, 1, 0\}$*. Unlike the DFA of Figure 1.3, this Mealy machine can be used to model the return values of the method calls.*

Register automata

DFAs and Mealy machines typically do not scale well if the domain of inputs, or the domain of data parameters for inputs, is large. A Mealy machine for a stack, for example, requires a number of states that is exponential in the size of the stack and the number of possible data parameters. This is already evident in our running example (Figure 1.4). Depending on the data parameters, seven different states are required for two consecutive calls of `push` followed by two calls of `pop`, despite that the behaviour is practically the same. The reason for this is that the semantics of the data parameters are modeled implicitly using states and transitions; inputs with different parameters are simply regarded as different inputs. A better solution is to

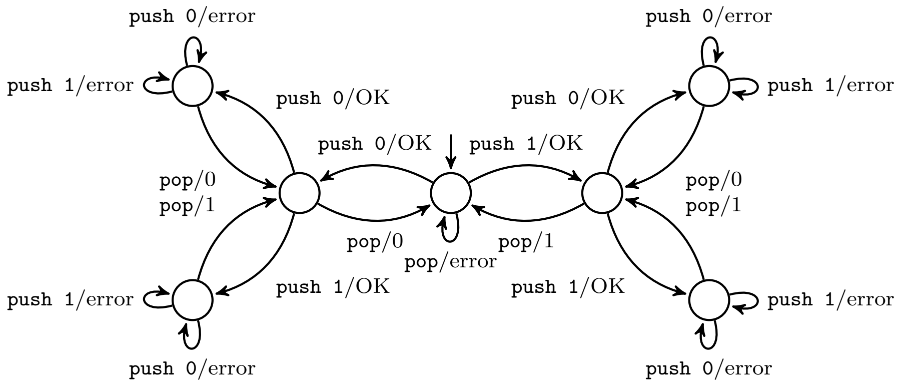


Figure 1.4: A Mealy machine for a stack of size 2

use a richer formalism that can model them more efficiently and exploit the resulting symmetries in the state space.

A *register automaton* is such a formalism. In this section, we describe register automata informally. We refer to Chapter 5 for a more formal treatment.

A register automaton can be seen as an automaton that is extended with a set of *registers* that can store data parameters. The values in these registers can then be used to express conditions over the transitions of the automaton, or *guards*. If the guard is satisfied, then the transition is fired, possibly storing the provided data parameter (this is called an *assignment*) and bringing the automaton from the current *location* to the next. In contrast to automata without memory, we speak of locations instead of states, because semantically, the *state* of a register automaton also comprises the values of the registers. Therefore, an infinite number of possible states can be modeled using a finite number of locations and registers.

Different flavours of register automata exist. In Chapter 5 we introduce a flavour that takes parameterized inputs and generates parameterized output. We restrict both inputs and outputs to a single parameter. Parameters may be stored in registers, and may be tested for equality with a stored value, or for inequality to all stored values. Output values can be equal to the stored values, or may be *fresh*.

Because it can describe input and output parameters, a register au-

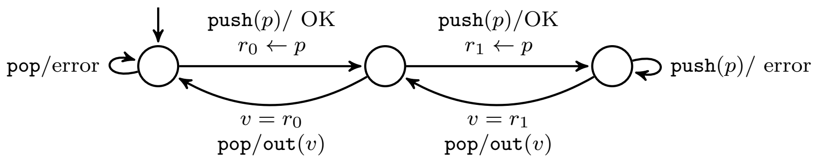


Figure 1.5: An RA for a stack of size 2

tomaton can be used to model control-specific input/output behaviour of a system and characterize relations between inputs and outputs. The data values are not examined to any extent, however. Only their (equality) relation to other data values is checked.

Example 1.7. *An RA can be used to fully describe the behaviour of a stack. Figure 1.5 shows such an RA for a stack of size 2. Here, guards are represented by expressions with the = sign, and assignments are represented by the ← operator. The RA has 2 registers (r_0 and r_1) that are initially empty.*

After receiving a `push` *method call parameterized with an element p, the RA stores this element in its first register. A subsequent parameter for a* `push` *call gets stored in the second register. If the RA is not in its initial location, a* `pop` *call returns the value v of a register as a parameter to the* `out` *output.*

Unlike the Mealy machine of Figure 1.4, this RA models the relations between input and output parameters. In addition, it represents the same behaviour more compactly.

1.1.3 Passive and Active Learning

Model learning approaches can be distinguished based on the way that information is presented to them. We make a distinction between *passive* and *active* learning.

In a *passive* setting, the learning algorithm is given a set of observations of the system's behaviour. As such, a typical information source for passive learning algorithms are *logs*. The problem that we are interested in is that of finding a (non-unique) smallest automaton that is consistent with these observations. Typically, the size of an automaton is measured by the number of states it contains.

In an *active* setting, the learning algorithm has the ability to interact with the system. As such, it can choose for which sequences of inputs it wants to observe the system's output. The problem is to find the smallest automaton that completely describes the system's behaviour. This means that the automaton is consistent with any observation of the system's behaviour that the algorithm has seen, or might ever see.

Each approach has its strengths and weaknesses. An obvious advantage for passive approaches is that they do not require access to the system, but only to a set of logs. As these logs often contain repeated observations, passive approaches can provide statistical information about the normal behaviour of the system. Such information can provide a basis for applications that active learning approaches are not suited for, such as *anomaly detection*.

The quality of a passively learned model is limited by the diversity of the given observations, however. If certain transitions in the automaton never occur, it is impossible to learn them. Active learning algorithms have the advantage that they can choose to explore these transitions. They can try out strange corner cases that never occur in practice. This may be useful to obtain a unique *fingerprint* of a system, for example. In our setting, this could be a set of accepted sequences or traces that is unique to the automaton that describes the system (and equivalent automata).

1.1.4 Learning in the Limit

The problem of learning automata from observations of their behaviour has been studied from a theoretical perspective for decades. The study was initiated by Edward Moore in 1956. In his classical paper he describes several problems, including that of inferring the state transition function of an unknown automaton and checking its correctness [104]. This problem has been studied in the subsequent years by many researchers, obtaining important theoretical results. An overview of this early work can be found in [137, Chapter 5].

In 1967, E. Mark Gold introduced the concept of *learning in the limit* [63]. In this setting, the problem of learning an automaton is viewed as a game involving a *learner* and a *teacher*. The goal for the learner is to infer the behaviour of an unknown automaton that is known by the teacher. Initially, the learner only knows the set of inputs for this automaton.

The game is seen as an infinite process. At each step, the learner is given an observation from the automaton's behaviour by the teacher. In response, the learner has to construct an automaton that is consistent with the observations so far. This automaton is called the *hypothesis*. The learner is said to have learned in the limit if after a finite number of steps the subsequent hypotheses are all the same, and are all equivalent to the teacher's automaton.

Learnability in the limit is a property of a *class* of automata, and is dependent on the *presentation mode* of the teacher. Presentation mode is a concept that can be used to distinguish different settings when learning DFAs and other flavours of automata that are used to distinguish between valid and invalid sequences of inputs. We make a distinction between the following presentation modes:

Text. In a *text* presentation, the learner is given only valid (accepted) sequences of inputs from the teacher's automaton.

Informant. In an *informant* presentation, the learner is given both valid and invalid sequences of inputs from the teacher's automaton, along with a label (accept or reject).

A class of automata is said to be learnable in the limit with respect to a presentation mode if there exists an algorithm with the following property:

> Given any automaton of the class and given any total enumeration of observations possible in the presentation mode, the automaton will be identified in the limit.

Learnability in the limit is an important concept in model learning. In a typical learning setting, a learner can not help but make mistakes sometimes. If she knows that the target automaton is learnable in the limit, however, she will at least be wrong only a finite number of times.

Only a small, insignificant subset of automata are learnable in the limit from text [63]. All the classes of automata that we are concerned with in this thesis are learnable in the limit from an informant, however. This includes Mealy machines and other automata that can be used to the input/output behaviour of software and hardware systems (the information provided by input-output traces is comparable to that of an informant). This means that there exist algorithms that can eventually learn the correct automaton when provided with the right set of observations. In the next section we present a framework for these algorithms.

1.1.5 The Minimally Adequate Teacher

In 1987, Dana Angluin showed that the behaviour of an automaton can efficiently be learned if it is presented by a so-called *minimally adequate teacher* (MAT) [11]. Similarly to learnability in the limit, MAT is a theoretical framework for learning automata in which the learning process can be seen as a game between a learner and a teacher. It is a framework for *active* learning, however, as the learner can ask two types of questions about the automaton:

Membership queries. *What is the automaton's response to this sequence of inputs?*

Equivalence queries. *Is the behaviour of the automaton equivalent to that of my hypothesis?* If not, the teacher provides a *counterexample*, which is a sequence of inputs for which a membership query and the hypothesis yield a different output.

The learner iteratively asks membership queries to construct an hypothesis. Once she has constructed one, she presents it to the teacher in an equivalence query. If the teacher's response is a counterexample, the learner uses membership queries to improve the hypothesis. This process continues until the learner's hypothesis is equivalent to the teacher's automaton.

A schematic overview of the MAT framework is shown in Figure 1.6.

1.1.6 The L^* Algorithm

In her seminal work, Angluin introduced an efficient algorithm for the MAT framework: the L^* algorithm. Variants of this algorithm are still widely used today. Let us present the variant introduced by Rivest and Shapire [119], because it is one of the easiest to understand. This variant of the algorithm

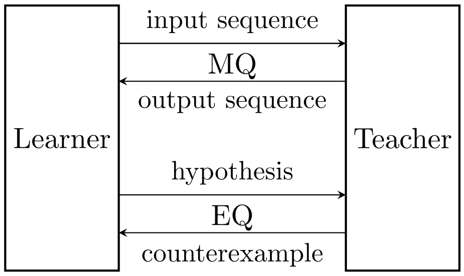


Figure 1.6: The minimally adequate teacher framework

is for learning DFAs, but it is applicable to other flavours of automata as well with minor modifications.

Let us assume that the teacher knows a DFA $A = (I, Q_A, q_A, \delta_A, F_A)$, and that we want to infer this DFA using the L^* algorithm. For the purpose of this example, let us introduce an *output function* $\lambda_A : Q_A \to \{\mathit{accept}, \mathit{reject}\}$ for A, such that:

$$\lambda_A(q) = \mathit{accept} \iff q \in F_A \quad \text{for all} \quad q \in Q_A$$

The core data structure of the L^* algorithm is called the *observation table*. The rows and columns of an observation table are labelled by prefixes and suffixes of input sequences respectively, and the cells are filled with the teacher's label in response to this input sequence (i.e. accept or reject). Membership queries are posed to fill the table. Some example observation tables are shown in Figure 1.7. These observation tables will be explained later.

The rows of an observation table are split in two groups. The top part is labeled by *access sequences*, and is used to identify the different states and the transitions required to reach these states. The bottom part is labeled by *one-input extensions*, and is used to map the remaining transitions. The columns are labeled by *separating sequences* that are used to discriminate rows in the top part of the table.

Formally, an observation table can be described by a triple (X, E, row), where:

– $X \subset I^*$ is a finite, prefix-closed set of *access sequences*, extended to include the one-input extensions $X \cup (X \cdot I)$,

- $E \subset I^*$ is a finite, suffix-closed set of *separating sequences*, and
- $\text{row} : (X \cup (X \cdot I) \to E) \to \{\textit{accept}, \textit{reject}\}$ is a function mapping these prefixes and suffixes to their output.

Initially, X and E contain the empty sequence ϵ.

A hypothesis can be constructed from the observation table if it is *closed*. This is the case if for each row labeled with a one-input extension there exists a row labeled by an access sequence that has an identical value in every column. If there is a row labeled with a one-input extension for which this is not the case, it is added as an access sequence. This continues until the table is closed.

Formally, the table is closed if for all $x \in X \cdot I$ there is a $y \in X$ such that $\text{row}(x) = \text{row}(y)$. If there is a row $x \in X \cdot I$ and there is no $y \in X$ such that $\text{row}(x) = \text{row}(y)$, then x is added to X.

Let $H = (I, Q_H, q_H, \delta_H, F_H)$ be a hypothesis DFA for A, and let $\lambda_H : Q_H \to \{\textit{accept}, \textit{reject}\}$ be an output function for H. H can be constructed from a closed observation table (X, E, row) as follows:

- I is given
- $Q_H = \{\text{row}(x) | x \in X\}$,
- $q_H = \text{row}(\epsilon)$,
- $\delta_H(\text{row}(x), a) = \text{row}(x \cdot a)$ for $a \in I$ and $x \in X$, and
- $\lambda_H(\text{row}(x)) = \text{row}(x)(\epsilon)$.

This hypothesis is presented to the teacher in an equivalence query. If the hypothesis DFA is not the same as the teacher's, an input sequence c is returned as a *counterexample*. This can be any input sequence for which the hypothesis and the teacher's DFA produce a different label, i.e. $\lambda_A(\delta_A(q_A, c)) \neq \lambda_H(\delta_H(q_H, c))$.

Several different methods for handling counterexamples exist (for an overview, see [131]). In the variant that we are concerned with, the counterexample and all of its suffixes are added to the table as suffixes (i.e. to E), and membership queries are used to fill the cells. This will violate the closedness of the observation table. Therefore, at least one new row will be added to the top part of the table, and the next hypothesis will have at least one extra state. Indeed, each subsequent hypothesis will have more states than the one before.

This procedure iterates until no counterexample can be found. At this point, H is equivalent to A.

Example 1.8. *Let us assume that we do not know the sequences of method calls to a stack of size 2 that are error free, but we have access to a teacher that knows the DFA of Figure 1.3.*

Our initial observation table is shown in Figure 1.7a. This table is not closed, because there are no rows labeled by an access sequence that are identical to row `pop`*. Therefore, we add* `pop` *to the top part of the table, and its one-input extensions to the bottom part of the table.*

The resulting table is shown in Figure 1.7b. This table is closed, and can be used to construct the hypothesis shown in Figure 1.8a.

A counterexample for this hypothesis is `push pop`*, because the hypothesis rejects this sequence, whilst the actual automaton accepts it. We handle the counterexample by adding it and all of its suffixes to the table as suffixes (i.e. to E). Membership queries are posed to fill the empty cells. The resulting observation table is shown in Figure 1.7c.*

This table is not closed, because there is no rows labeled by an access sequence that is identical to row `push`*. Therefore, we add* `push` *to the top part of the table, and its one-input extensions to the bottom part of the table.*

The resulting table is shown in Figure 1.7d. Again, this table is not closed because of row `push push`*. After handling this row we obtain the closed observation table shown in Figure 1.7e. The corresponding hypothesis for this table is shown in Figure 1.8b. Observe that this is the same DFA as that of Figure 1.3. Therefore, we have successfully learned a model for sequences of method calls to a stack of size 2 that are error free.*

1.1.7 Conformance Testing

In Section 1.1.4, we have established that it is possible to learn a correct model of a real-world system, if its behaviour can be characterized by an automaton. Moreover, in Section 1.1.5 we have sketched a framework for doing so, and in Section 1.1.6 we have introduced an efficient algorithm for this framework. This framework relies on the expertise of a teacher, however,

		E
		ϵ
X	ϵ	*accept*
$X \cdot I$	`push`	*accept*
	`pop`	*reject*

(a) Initial, unclosed table (row `pop`)

		E
		ϵ
X	ϵ	*accept*
	`pop`	*reject*
$X \cdot I$	`push`	*accept*
	`pop push`	*reject*
	`pop pop`	*reject*

(b) Table for first hypothesis

			E	
		ϵ	`push pop`	`pop`
X	ϵ	*accept*	*accept*	*reject*
	`pop`	*reject*	*reject*	*reject*
$X \cdot I$	`push`	*accept*	*accept*	*accept*
	`pop push`	*reject*	*reject*	*reject*
	`pop pop`	*reject*	*reject*	*reject*

(c) Unclosed table after counterexample `push pop` (row `push`)

			E	
		ϵ	`push pop`	`pop`
X	ϵ	*accept*	*accept*	*reject*
	`pop`	*reject*	*reject*	*reject*
	`push`	*accept*	*accept*	*accept*
$X \cdot I$	`pop push`	*reject*	*reject*	*reject*
	`pop pop`	*reject*	*reject*	*reject*
	`push push`	*accept*	*reject*	*accept*
	`push pop`	*accept*	*accept*	*reject*

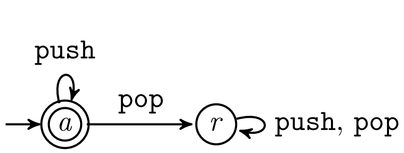
(d) Unclosed table (row `push push`)

Figure 1.7: Observation tables for the DFA of Figure 1.3

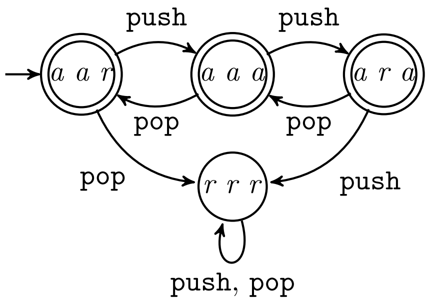
			E	
		ϵ	`push pop`	`pop`
X	ϵ	*accept*	*accept*	*reject*
	`pop`	*reject*	*reject*	*reject*
	`push`	*accept*	*accept*	*accept*
	`push push`	*accept*	*reject*	*accept*
$X \cdot I$	`pop push`	*reject*	*reject*	*reject*
	`pop pop`	*reject*	*reject*	*reject*
	`push pop`	*accept*	*accept*	*reject*
	`push push push`	*reject*	*reject*	*reject*
	`push push pop`	*accept*	*accept*	*accept*

(e) Final, closed table

Figure 1.7: Observation tables for the DFA of Figure 1.3 (cont.)



(a) First hypothesis (see Fig. 1.7b)



(b) Final hypothesis (see Fig. 1.7e)

Figure 1.8: Hypotheses for the DFA of Figure 1.3. The states are labeled with a mapping from suffixes in the observation table to the outputs for that state (a = *accept*, r = *reject*).

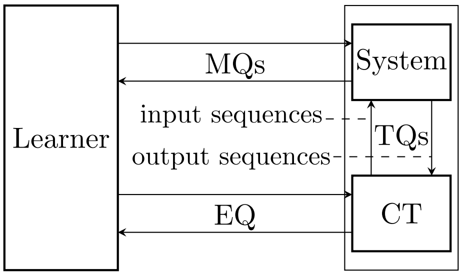


Figure 1.9: Learning in practice using conformance tesing (CT)

and in most practical scenarios such a teacher does not exist. Membership queries can be answered by interacting with the system, but there is no trivial way of implementing equivalence queries. Therefore, researchers were unaware of the practical avail of this work for decades.

Only in more recent years Peled et al. made the observation that equivalence queries can be approximated using a technique called *conformance testing* [114]. In the context of model learning, the goal of conformance testing is to establish an equivalence relation between the current hypothesis and the system. This is done by posing a set of so-called *test queries* to the system. In a test query, similarly to a membership query, the learner asks for the system's response to a sequence of inputs. If the system's response is the same as the predicted response (by the hypothesis) for all test queries, then the hypothesis is assumed to be equivalent to the target. Otherwise, if there is a test for which the target and the hypothesis produce different outputs, then this input sequence can be used as a counterexample.

A schematic overview of the MAT framework in practice (using conformance testing) is shown in Figure 1.9.

One of the main advantages of using conformance testing is that it can distinguish the hypothesis from all other automata of size at most m, where m is a user-selected bound on the number of states. This means that if we know a bound m for the size of the system we learn, we are guaranteed to find a counterexample if there exists one. For an overview of some m-complete conformance testing methods, we refer to [47].

Complete conformance testing methods require the following information:

1. A set of *access sequences*, possibly extended with their one-input

extensions to obtain a *transition cover* set.

2. A *traversal set* that contains all input sequences of length $l = m - n + 1$, where m is the (typically unknown) number of states of the system, and n is the number of states of our hypothesis.

3. A means of pairwise distinguishing all states of our hypothesis, such as set of *separating sequences* for all pairs of states.

A test suite is then constructed by taking the product of these sets, or subsets of these sets.

The difference between various complete conformance testing methods is how states are distinguished (i.e. the last part). Finding such separating sequences for all pairs of states of an automaton is a classic problem in automata theory. When we are learning a system using the L^* algorithm, we can use the suffixes of the observation table for this purpose.

Example 1.9. *Table 1.1 shows the access sequences, traversal set and separating sequences for the hypothesis DFA of Figure 1.8. A 4-complete test set can be obtained from these sets by taking the product of these sets. As such, the complete test set contains 26 input sequences (if we do not remove duplicates). In comparison, a 3-complete test set would have only 10 input sequences, while a 5-complete test set would have 58. Indeed, the size of a complete test set grows exponentially.*

1.2 Contributions

The remaining chapters of this thesis contain the research papers that the author has contributed to. Each of these chapters builds upon one or more of the concepts that have been introduced in this introduction, and can be read independently of the others. The naming and notation of these concepts might differ, however, as the papers were intended for different audiences. In this section we outline the contributions of each of these chapters.

Chapter 2: Minimal Separating Sequences for All Pairs of States

Finding *minimal* separating sequences for all pairs of states of an automaton is a classic problem in automata theory. As we have explained in

Table 1.1: A 4-complete test set for the hypothesis of Figure 1.8

access sequences	traversal set	separating sequences
ϵ	ϵ	ϵ
`pop`	`push`	
	`pop`	
	`push push`	
	`push pop`	
	`push push push`	
	`push push pop`	
	`push pop push`	
	`push pop pop`	
	`pop push push`	
	`pop push pop`	
	`pop pop push`	
	`pop pop pop`	

Section 1.1.7, these sequences play a central role in conformance testing methods. One way of obtaining separating sequences is through *partition refinement*. In 1956, Edward Moore already outlined a partition refinement algorithm that constructs a set of minimal separating sequences in $\mathcal{O}(mn)$ time, where m is the number of transitions and n is the number of states of the automaton [104].

In this chapter, we present an improved algorithm for obtaining minimal separating sequences based on the famous partition refinement algorithm of Hopcroft [72] that runs in $\mathcal{O}(m \log n)$ time. The theoretical complexity of our algorithm is empirically verified and compared to the traditional algorithm. We found our algorithm to be faster in practice.

This chapter is based on the following publication:

R. Smetsers, J. Moerman, and D. Jansen. Minimal separating sequences for all pairs of states. In *Proceedings LATA*, volume 9618 of *LNCS*, pages 181–193. Springer, 2016.

The author has presented this work at the 10th International Conference on Language and Automata Theory and Applications (LATA) in Prague, Czech Republic on Tuesday, March 15 2016.

Chapter 3: Bigger is Not Always Better

The L^* algorithm is characterized by its iterative construction of hypotheses. Each subsequent hypothesis has more states than the previous one. In this chapter, we show that a bigger model is not always better. We show that the minimal length of a counterexample that distinguishes a hypothesis from the system may decrease, and we present a modification of the L^* algorithm that ensures that this is not the case. As a result, the distance to the system never increases in a corresponding ultrametric. Preliminary experimental evidence suggests that our algorithm speeds up learning in practical applications by reducing the number of equivalence queries.

This chapter is based on the following publication:

> R. Smetsers, M. Volpato, F. Vaandrager, and S. Verwer. Bigger is not always better: on the quality of hypotheses in active automata learning. In *Proceedings ICGI*, volume 34 of *JMLR: W&CP*, pages 167–181, 2014.

The author has presented this work at the 12th International Conference on Grammatical Inference (ICGI) in Kyoto, Japan on Friday, September 19 2014.

Chapter 4: Enhancing Automata Learning by Log-Based Metrics

In Chapter 3 we introduce the relevance of ultrametrics in the context of model learning presented in Chapter 3. In this chapter we study a general class of distance metrics for deterministic Mealy machines. Our metrics are induced by weight functions that specify the relative importance of input sequences. By choosing an appropriate weight function we may fine-tune a metric so that it captures some intuitive notion of quality for hypotheses.

In particular, we present a metric that is based on the minimal number of inputs that must be provided to obtain a counterexample, starting from states that can be reached by a given set of logs.

For any weight function, we may boost the performance of existing model learning algorithms by introducing an extra component, which we call the *Comparator*. Preliminary experiments show that the Comparator yields a significant reduction of the number of inputs required to learn correct models. Moreover, by generalising the result of Chapter 3, we show that the quality of hypotheses that are generated by the Comparator never decreases.

This chapter is based on the following publication:

> P. van den Bos, R. Smetsers, and F. Vaandrager. Enhancing automata learning by log-based metrics. In *Proceedings IFM*, volume 9681 of *LNCS*, pages 295–310. Springer, 2016.

Petra van den Bos has presented this work at the 12th International Conference on Integrated Formal Methods in Reykjavik, Iceland on Thursday, June 2 2016.

Chapter 5: Model Learning as an SMT Problem

In this chapter we explore an approach to model learning that is based on using *satisfiability modulo theories* (SMT) solvers. SMT is the problem of deciding if there exists an assignment to a logic formula that makes it true. We explain how the different automata formalisms introduced in Section 1.1.2 and observations of their behaviour can be encoded as logic formulas. An SMT solver is then tasked with finding an assignment for such a formula, from which we can extract an automaton of minimal size.

We provide an implementation of this approach which we use to conduct experiments on a series of benchmarks. These experiments address both the scalability of the approach and its performance relative to existing active learning tools.

This chapter is based on the following publication:

> R. Smetsers, Paul Fiterău-Broştean, and F. Vaandrager. Model learning as a satisfiability modulo theories problem. In *Proceedings LATA*, volume 10792 of *LNCS*, in press. Springer, 2018.

Frits Vaandrager will present this work at the 12th International Conference on Language and Automata Theory and Applications in Bar-Ilan, Israel between April 9 and 11, 2018.

The author has presented a preliminary version of this chapter at the 1st Workshop on Learning and Automata (LearnAut) in Reykjavik, Iceland on Monday, June 19 2017. This version is available on arXiv (arXiv:1705.10639).

Chapter 6: Complementing Model Learning with Fuzzing

An ongoing challenge for learning algorithms formulated in the MAT framework is to efficiently implement equivalence queries. The typical approach of using conformance testing for this purpose (as outlined in Section 1.1.7) has some notable drawbacks. First, it is hard (or even impossible) in practice to determine an upper-bound on the number of states of the system's automaton. Second, it is known that conformance testing becomes exponentially more expensive for higher values of this bound. As such, the learner might incorrectly assume that its hypothesis is correct.

In this chapter, we compare and combine conformance testing and *mutation-based fuzzing* methods for obtaining counterexamples. In essence, *fuzzers* are programs that apply a test (i.e. input sequence) to a target program, and then iteratively modify this sequence to monitor whether or not something interesting happens (e.g. crash, different output, increased code coverage).

We have used this approach in the Rigorous Examination of Reactive Systems (RERS) challenge of 2016 with good results, winning most of the competition's categories. This leads us to believe that testing and fuzzing are orthogonal and complementary approaches in the context of model learning. In this chapter, we therefore describe our experimental setup for RERS in detail and we describe possible ways of combining learning and fuzzing.

The author has presented this chapter at the Rigorous Examination of Reactive Systems workshop at the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation on Sunday, October 9 2016. The paper is currently available on aXiv (arXiv:1611.02429), and has been invited for a special issue on RERS in the International Journal on Software Tools for Technology Transfer (STTT).

Chapter 7: Protocol Message Format Inference and its Applications in Security

A promising application of model learning is in the area of *protocol inference*. Protocol inference refers to some automated form of reverse engineering the workings of a communication protocol. This can be useful for security analysis in different ways. It can be used to reverse-engineer unknown protocols, to detect security flaws in implementations of known protocols, to fingerprint implementations, or to detect anomalies in protocol usage, for example.

A prerequisite for using model learning in this area is that the protocol's so-called *message format* (i.e. input format) is known. In this chapter, we give an overview of tools and techniques for inferring the protocol message format, and their applications in security. It was observed by Bossert and Guilhéry that there is a huge difference between the academic and the applied world in the field of protocol inference for security applications [24]. This chapter aims to bridge that gap, and propagate further research in the area.

This chapter has not been published or presented anywhere.

1.3 Related Work

Before we give a detailed outline of the related work for the contributions of this thesis, let us refer to some excellent publications that give a more complete overview of the different approaches for model learning. First, Cook and Wolf present a general framework for passive learning and give an excellent introduction to early techniques for passively learning the behaviour of software systems [41]. Second, Vaandrager gives a concise overview of the key advancements and current challenges in active model learning [140]. Finally, De la Higuera has written a complete reference work for the different algorithms that have been proposed for learning automata (and other formalisms), independent of the application area [71].

1.3.1 Passive Learning Algorithms

The problem of learning a minimal size automaton model from a set of observations can be very hard. It is the optimization variant of the problem of finding a consistent automaton of a fixed size, which has been shown to be NP-complete [64], and in-approximable [116]. Therefore, most algorithms for passive model learning are based on a greedy technique known as *state merging*. State merging algorithms start with a tree-shaped automaton that exactly encodes the set of observations. The algorithm then iterates through the following steps:

Select two states q, q' with similar and consistent futures.

Merge q and q': all transitions from (resp. to) q' are added as transitions from (resp. to) q, and q' is removed.

Determinize the automaton: iteratively merge all target states of transitions from merged states.

This process continues for as long as valid merges are possible.

Merge validity can be defined in different ways, and this is where the many different state merging algorithms differ. The following algorithms have had the biggest impact on the field:

***k*-tails** is a variant of state merging that computes the consistency of futures up to a given length k [20].

RPNI only disallows merges that combine an accepting state with a rejecting state, either directly or during determinization [112].

RPNI2 is an incremental version of RPNI that can be used in active learning frameworks [53].

EDSM does the same as RPNI, but in addition prioritizes merges for more similar state pairs [90].

ALERGIA only merges states with similar outing transition frequencies, and learns a probabilistic automaton [31].

MDI also learns a probabilistic automaton, but uses Kullback-Leibler divergence to compute state similarity and consistency [134].

EXBAR performs an iterative deepening search around EDSM [89].

BEAM uses a search method guided by the information-theoretic *Occam's razor* principle to infer probabilistic automata [118].

DFASAT first performs merges that combine the most accepting states, and provides a search routine using a satisfiability solver [69].

State merging algorithms are typically designed to learn in the limit from polynomial time and data [112], and have formed the basis for the winning contributions to early model learning competitions: Abbadingo in 1998 [90] and STAMINA in 2010 [145].

A major downside of state merging is that, if errors are made early on in the state merging process, they are compounded by future merges. As such, the accuracy of the final result is highly dependent on the set of observations that is available.

State merging is typically used in the context of DFA or probabilistic automata, but has been applied to other formalisms as well, such as Moore machines [61] and timed automata [143].

The alternative to state merging is to express the problem of learning a minimal consistent automaton from observations as a *constraint satisfaction problem* (CSP), and use a *constraint solver* to find a solution. The approach that we take in Chapter 5 (satisfiability modulo theories) can be thought of as such a constraint satisfaction problem.

Coste and Nicholas were the first to recognize that the problem of learning a DFA from observations can be reduced to a graph coloring problem [42]. In [68, 69], Heule and Verwer show that this reduction can be encoded in propositional logic. *Satisfiability*, or SAT, is the constraint satisfaction problem of deciding if there exists an assignment to a propositional logic formula that makes it true. As such, the tool of Heule and Verwer uses a *SAT solver* to find a minimal consistent automaton.

The encoding of Heule and Verwer was adapted for learning extended finite automata from *test scenarios* by Ulyantsev et al. [139]. *Test scenarios* are sequences of elements that consist of an input, a guard condition over this input and one or more outputs. As such, test scenarios contain more information than the (parameterized) input sequences that we learn from in Chapter 5.

For the problem of learning minimal consistent automata from observations, an encoding in propositional logic might not be the right choice, as the encoding might become complex very quickly, and therefore cumbersome to maintain and adapt. A better alternative is to express it in a richer logic. This was recognised by Bruynooghe et al. [28], who express the encoding by Heule and Verwer in a predicate logic, and by Chivilikhin et al. [38], who express the aforementioned problem of learning extended finite automata from test scenarios in a higher level CSP language.

1.3.2 Passive Learning Applications

Several studies have applied passive learning to discover some of the temporal and data dependence relationships of a software system's protocol.

Ammons et al. have applied this technique for X11 programs, for example [10]. Their input sequences consist of function calls and their attributes (i.e. parameters). Because the interface of such a program is typically open-source and well documented, no message format reverse engineering is applied here. Instead, method calls are instrumented, and several preprocessing steps are taken to make the domain of the message attributes finite and group similar sequences. The resulting input sequences are called *scenarios*. These scenarios are in turn used to learn a probabilistic automaton using the BEAM algorithm [118].

The approach described above was applied with mixed success to a handful of X11 programs. Scenarios were obtained from the X library calls and callbacks from and to these programs. These scenarios were then verified to comply to the Inter-Client Conventions Manual (ICCM), which is a standard for interoperability between X Window System clients of the same X server. Out of the 16 programs analysed, five violated a rule in the ICCM. Two of these violations to the standard were caused by bugs in the implementation.

In 2011, Lee et al. observed that sequences of method calls observed from a system often deal with multiple independent receivers [91]. One such example is an sequence of interleaved calls to a library from two concurrent threads in a program. It is hard to learn something useful from interleaved sequences, because there are many interleavings possible. Moreover, there might be no semantic relation between the different recipients. The authors observe that different recipients can be distinguished based on the values of certain message parameters. They propose jMiner, an effective and general-purpose passive learning approach for inferring automata from interleaved message traces.

jMiner works by first partitioning the sequence in a set of independent ones, based on the parameters of the messages. For this, it uses a message format specification that it infers from the source code, package name, and unit test cases. The independent sessions are then used to learn a FSM using the same off-the-shelf passive learner as Ammons et al. used. The authors have successfully applied jMiner to a set of interleaved message sequences from four packages of OpenJDK.

Yang et al. have applied passive learning for dynamically inferring functional specifications from method calls [151, 152]. These specifications can be seen as invariants for how the interaction with the interface behaves. A functional specification for a *mutex*, for example, might be that "*mutex.acquire(X)* is always followed by *mutex.release(X)*". The inference engine of their Perracotta tool uses heuristics to generalize these rules into regular expressions. These regular expressions can be represented by a DFA.

Although the previously described tools can often be used to accurately describe behavior of a specific system, they sometimes fail to capture crucial dependencies between input parameters. As we have outlined in this introduction, automata that model the control flow of a system typically only provide a partial view of that protocol's behaviour. In practice, behaviour is often the result of interplay between the input sequences (as described by the automaton), and the values of the parameters for these messages. Therefore, most recent work focusses on learning both these facets of behaviour.

Indeed, these facets require the automaton to operate on an underlying memory, and have its transitions annotated by guards on the memory. We have already seen a formalism that can do this: the register automaton.

Walkinshaw et al. have recently proposed a passive technique for learning a flavour of register automata from method calls [146]. The technique works by combining previously mentioned passive learning techniques—which infer the control flow from inputs—with a component that relates the input sequences to the data state of the system. The latter process is known in general as *data classifier inference*, and refers to a range of techniques that that map possible values for parameters to a particular class. In the case of model learning, we are interested in the next input that will follow. Therefore this is the class that the authors try to predict in their application of data classifier inference.

There exist a huge number of classifiers that can solve this task. In their experiments, Walkinshaw et al. observe that the choice of classifier ultimately depends on the application area and the context in which the automaton is used. Therefore, the authors have made an effort to enable the use of an arbitrary classifier. In their reference implementation, called Mint, they use the (fifty or so) classifiers that are in the WEKA library [66].

The approach described above was applied with mixed success to a communications protocol that allocates frequencies to mobile phones, an implementation of a 'resource locker', and three Java SDK classes. The authors have identified two characteristics for an ideal application scenario of the technique.

– From a data dependency point of view, an input should contain only parameters that have a direct bearing on the subsequent behaviour, and the number of these parameters would ideally be low for each input.

– From a control flow point of view, the number of elements that could possibly succeed a given input should ideally be low (to prevent incorrect data-based classification), however all of these possible sequences should be represented in the set of observations.

1.3.3 Active Learning Algorithms

In Angluin's L^* algorithm an observation table typically contains a lot of duplicate information. As a result, it can grow in size quickly. In recent years, many improvements and variations on Angluin's original algorithm have been presented. We refer to Balcazar et al. for an overview of early work [15], and to Steffen et al. for a more recent self-contained description and overview [131]. Optimisations for large alphabets are discussed by Irfan et al. [81], and practical optimisations that improve both the number of queries and time required to answer them are given by Bauer et al. [16] and

Irfan et al. [80]. Adaptations of the algorithm that handle counterexamples differently are given by Rivest and Shapire [120] and Shahbaz and Groz [124].

The culmination of this development is the TTT algorithm by Isberner et al. [83]. In comparison to the earlier algorithms, the TTT algorithm is particularly well suited for handling long counterexamples. It does so by maintaining a so-called *spanning tree* and a so-called *discrimination tree.* The spanning tree keeps track of the message prefixes that lead to unique states in the hypothesis. These states of the hypothesis correspond to leaves of the *discrimination tree*, whose inner nodes are labelled with distinguishing suffixes, and whose transitions are labelled with outputs. For a node labelled with suffix x, the subtree reached by transition labelled with output o contains the states for which o is the last output in response to x. Hence, for every pair of states, a discriminator can be obtained by looking at the label of the *least common ancestor* of the corresponding leaves. This can be used to determine the transitions in the hypothesis.

The way that the TTT algorithm handles counterexamples is based on the observation by Rivest and Shapire that a counterexample can be decomposed to point out the transition in the hypothesis that is incorrect [120]. In the discrimination tree, the leaf corresponding to the source state of this transition is replaced with an inner node that is *temporarily* labelled by a suffix from the counterexample. A technique known as *discriminator finalization* is then applied to construct the subtree of this newly created inner node, and possibly obtain a shorter suffix. For a more in-depth description of discriminator finalization and the TTT algorithm in general, we refer to [83].

The aforementioned approaches can be used for learning the control flow of a system. In some situations, however, it might be useful to learn the data flow of the system as well. In these situations, inputs are typically drawn from an infinite domain and automata frameworks over infinite alphabets (such as register automata) become natural models. In recent years, several different approaches for learning over infinite (or large) alphabets have been developed.

Two approaches have been proposed for learning register automata. The first approach aims to infer a so-called *Nerode equivalence.* Such an equivalence provides a necessary and sufficient condition for distinguishing the automaton's states. This approach has been implemented as part of the

LearnLib tool [76], and its RALib extension [33]. For an overview of this line of work we refer to [32].

The second approach uses counterexample-guided abstraction refinement to automatically construct an appropriate *mapper*. A mapper is a component in the learning process that abstracts a large number of possible (parameterized) inputs into a limited number of abstract ones in a history-dependent manner, and vice versa [5]. The concept of a mapper has been implemented in the Tomte tool [3]. For an overview of this line of work we refer to [3] and [1].

Moerman et al. present an abstract approach for learning *nominal automata*, a formalism that is similar to register automata [103]. Nominal automata can be used to represent *nominal sets*, which are infinite sets equipped with symmetries. The authors present a generalization of the L^* algorithm that follows a generic pattern for transporting computation models from (traditional) finite sets to nominal sets, which leads to simple correctness proofs and opens the door to further generalizations. In addition, they present a variant for nondeterministic nominal automata, which are strictly more expressive.

Mens et al. take a more direct approach to the problem of learning over large or infinite alphabets [100]. They use an automaton framework known as *symbolic automata*, in which transitions are labeled by elements of a finite partition of the set of inputs. This way, symbolic automata can represent large alphabets much more succinctly than standard automata without memory, such as a DFA. The size of a DFA grows linearly with the size of the set of inputs and so does the complexity of active learning algorithms such as L^*. The authors present a variant of L^* for symbolic automata whose complexity is independent of the set of inputs.

Approaches that combine passive and active learning techniques are of particular interest to us, as Chapter 4 and Chapter 5 touch upon this subject. Walkinshaw et al. present such a hybrid approach for learning the behaviour of software systems [144]. They present an active learning framework that is similar to the one that we introduce in Chapter 5, in which no membership queries are used. Instead it iteratively runs a partial conformance testing method and a state merging algorithm that has been shown to be effective for sparse samples of observations (EDSM, see Section 1.3.1 and [90]).

A major strength of such an approach is that it does not rely on systematic (complete) exploration of the state space. As such, it can arrive at a reasonable hypothesis after a relatively small number of observations.

A downside of the approach is that, if errors are made early on in the state-merging process, they are compounded by future merges. As such, the accuracy of the final result is highly dependent on the testing technique that is used, and relies on a solid set of input sequences that prevents invalid merges from happening. The approach that we present in Chapter 5 overcomes this problem because it does not propagate errors that it makes when there are insufficient observations available.

1.3.4 Active Learning Applications

Active model learning has been successfully applied for discovering the behaviour of software systems on numerous occasions. In this section, we give an overview of some of the most notable studies.

In 2010, Cho et al. were the first to demonstrate how active model learning can be used for analysing *botnets* [40]. They used an adaptation of the L^* algorithm for learning the MegaD Command and Control (C&C) protocol. MegaD is a botnet that at its prime accounted for 32% of global spam [130]. In their analysis of the protocol they show how to identify its weakest links and design flaws. Besides, they were able to prove the existence of unobservable back-channels between botnet servers, without having access to these servers.

By leveraging properties specific to most network protocols, the heuristics introduced by Cho et al. allow for learning state machines in a realistic high-latency network setting. Compared to the original L^* algorithm, the time to learn the MegaD C&C protocol was reduced from days to hours with these heuristics. Primarily, the authors observed that communication protocols typically only accept a subset of all inputs at most times. This allowed them to greatly reduce the number of queries asked in the (active) learning process. Also, they had great success using *parallel processing* and *caching* of queries.

Also in 2010, Aarts et al. were the first to apply model learning for the analysis of *smart cards* (i.e. a card that has a chip) [7]. They used the L^* algorithm for analysis of electronic passports.

Later, they have used a similar approach for learning models of bank cards that support the EMV protocol [2]. Although they did not find any flaws in these cards, their analysis does reveal differences in the implementation between cards that are supposed to implement the same protocol.

To be able to analyse the *e.dentifier2*, a USB connected bank card reader, Chalupar et al. make use of a Lego robot in order to perform physical interactions with the device in the learning process [36]. As the USB implementation in the original system does not always provide reliable results, they make use of majority voting to determine the output.

Fiterău-Broştean et al. apply model learning with mappers to the TCP network protocol [55, 54]. They show that different implementations of TCP in Windows 8 and Ubuntu induce different models, which allows for fingerprinting of these implementations. Inspection of the learned models reveals that both Windows 8 and Ubuntu violate RFC 793 – the standard that describes the TCP protocol.

De Ruiter et al. use model learning in their analysis of nine different TLS implementations [122]. They found security related flaws in three of these implementations.

Fiterău-Broştean et al. apply model learning on three SSH implementations to infer automata models, and then use *model checking* to verify that these models satisfy basic security properties and conform to the RFCs [56]. Their analysis showed that all tested SSH server models satisfy the stated security properties. They did uncovered several violations of the standard, however, which may allow for fingerprinting of the different implementations.

Model learning has been applied to *industrial control software* on several occasions.

Smeenk et al. use model learning to validate the correctness of a software component that is used in printers and copiers of Océ [127]. Their main challenge was that traditional conformance testing methods were unable to find counterexamples for some hypotheses. They therefore implemented an extension of the algorithm of Lee and Yannakakis for computing an adaptive distinguishing sequence [92]. Even when an adaptive distinguishing sequence does not exist, Lee and Yannakakis' algorithm produces an adaptive sequence that 'almost' identifies states. In combination with a standard algorithm for computing separating sequences for pairs of states, the authors managed to verify states with on average 3 test queries. Altogether, they needed around 60 million queries to learn a model of the ESM with 77 inputs and 3.410 states.

Schuts et al. use model learning and model checking to compare a legacy implementation to a new implementation of a component at Philips Healthcare [123]. Instead of comparing the two implementations via their internal structure, they check the equivalence of their behaviour. First they use model learning to construct a model for both the legacy implementation and the new one. Then, they use model checking to see if the learned models are equivalent. This way, they found issues in both the legacy implementation and the new one. After solving these issues, model learning helped to increase confidence that the two implementations behave the same.

Chapter 2

Minimal Separating Sequences for All Pairs of States

Rick Smetsers, Joshua Moerman, and David N. Jansen

Abstract

Finding minimal separating sequences for all pairs of inequivalent states in a finite state machine is a classic problem in automata theory. Sets of minimal separating sequences, for instance, play a central role in many conformance testing methods. Moore has already outlined a partition refinement algorithm that constructs such a set of sequences in $\mathcal{O}(mn)$ time, where m is the number of transitions and n is the number of states. In this chapter, we present an improved algorithm based on the minimization algorithm of Hopcroft that runs in $\mathcal{O}(m \log n)$ time. The efficiency of our algorithm is empirically verified and compared to the traditional algorithm.

2.1 Introduction

In diverse areas of computer science and engineering, systems can be modelled by *finite state machines* (FSMs). One of the cornerstones of automata theory is minimization of such machines (and many variation thereof). In this process one obtains an equivalent minimal FSM, where states are different if and only if they have different behaviour. The first to develop an algorithm for minimization was Moore [104]. His algorithm has a time complexity of $\mathcal{O}(mn)$, where m is the number of transitions, and n is the number of states of the FSM. Later, Hopcroft improved this bound to $\mathcal{O}(m \log n)$ [72].

Minimization algorithms can be used as a framework for deriving a set of *separating sequences* that show *why* states are inequivalent. The separating sequences in Moore's framework are of minimal length [62]. Obtaining minimal separating sequences in Hopcroft's framework, however, is a non-trivial task. In this chapter, we present an algorithm for finding such minimal separating sequences for all pairs of inequivalent states of a FSM in $\mathcal{O}(m \log n)$ time.

Coincidentally, Bonchi and Pous recently introduced a new algorithm for the equally fundamental problem of proving equivalence of states in non-deterministic automata [22]. As both their and our work demonstrate, even classical problems in automata theory can still offer surprising research opportunities. Moreover, new ideas for well-studied problems may lead to algorithmic improvements that are of practical importance in a variety of applications.

One such application for our work is in *conformance testing*. Here, the goal is to test if a black box implementation of a system is functioning as described by a given FSM. It consists of applying sequences of inputs to the implementation, and comparing the output of the system to the output prescribed by the FSM. Minimal separating sequences are used in many test generation methods [47]. Therefore, our algorithm can be used to improve these methods.

2.2 Preliminaries

We define a FSM as a Mealy machine $M = (I, O, S, \delta, \lambda)$, where I, O and S are finite sets of *inputs*, *outputs* and *states* respectively, $\delta : S \times I \to S$ is a *transition function* and $\lambda : S \times I \to O$ is an *output function.* The functions δ and λ are naturally extended to $\delta : S \times I^* \to S$ and $\lambda : S \times I^* \to O^*$. Moreover, given a set of states $S' \subseteq S$ and a sequence $x \in I^*$, we define $\delta(S', x) = \{\delta(s, x) | s \in S'\}$ and $\lambda(S', x) = \{\lambda(s, x) | s \in S'\}$. The *inverse transition function* $\delta^{-1} : S \times I \to \mathcal{P}(S)$ is defined as $\delta^{-1}(s, a) = \{t \in S | \delta(t, a) = s\}$.

Observe that Mealy machines are deterministic and input-enabled (i.e. complete) by definition. The initial state is not specified because it is of no importance in what follows. For the remainder of this chapter we fix a machine $M = (I, O, S, \delta, \lambda)$. We use n to denote its number of states, i.e. $n = |S|$, and m to denote its number of transitions, i.e. $m = |S| \cdot |I|$.

Definition 2.1. *States s and t are* equivalent *if $\lambda(s, x) = \lambda(t, x)$ for all x in I^*.*

We are interested in the case where s and t are not equivalent, i.e. *inequivalent.* If all pairs of distinct states of a machine M are inequivalent, then M is *minimal.* An example of a minimal FSM is given in Figure 2.1.

Definition 2.2. *A* separating sequence *for states s and t in s is a sequence $x \in i^*$ such that $\lambda(s, x) \neq \lambda(t, x)$. We say x is* minimal *if $|y| \geq |x|$ for all separating sequences y for s and t.*

A separating sequence always exists if two states are inequivalent, and there might be multiple minimal separating sequences. Our goal is to obtain minimal separating sequences for all pairs of inequivalent states of M.

2.2.1 Partition Refinement

In this section we will discuss the basics of minimization. Both Moore's algorithm and Hopcroft's algorithm work by means of partition refinement. A similar treatment (for DFAs) is given in [65].

A *partition* P of S is a set of pairwise disjoint non-empty subsets of S whose union is exactly S. Elements in P are called *blocks.* If P and P'

are partitions of S, then P' is a *refinement* of P if every block of P' is contained in a block of P. A partition refinement algorithm constructs the finest partition under some constraint. In our context the constraint is that equivalent states belong to the same block.

Definition 2.3. *A partition is* valid *if equivalent states are in the same block.*

Partition refinement algorithms for FSMs start with the trivial partition $P = \{S\}$, and iteratively refine P until it is the finest valid partition (where all states in a block are equivalent). The blocks of such a *complete* partition form the states of the minimized FSM, whose transition and output functions are well-defined because states in the same block are equivalent.

Let B be a block and a be an input. There are two possible reasons to split B (and hence refine the partition). First, we can *split B with respect to output after a* if the set $\lambda(B, a)$ contains more than one output. Second, we can *split B with respect to the state after a* if there is no single block B' containing the set $\delta(B, a)$. In both cases it is obvious what the new blocks are: in the first case each output in $\lambda(B, a)$ defines a new block, in the second case each block containing a state in $\delta(B, a)$ defines a new block. Both types of refinement preserve validity.

Partition refinement algorithms for FSMs first perform splits w.r.t. output, until there are no such splits to be performed. This is precisely the case when the partition is *acceptable*.

Definition 2.4. *A partition is* acceptable *if for all pairs s, t of states contained in the same block and for all inputs a in I, $\lambda(s, a) = \lambda(t, a)$.*

Any refinement of an acceptable partition is again acceptable. The algorithm continues performing splits w.r.t. state, until no such splits can be performed. This is exactly the case when the partition is stable.

Definition 2.5. *A partition is* stable *if it is acceptable and for any input a in I and states s and t that are in the same block, states $\delta(s, a)$ and $\delta(t, a)$ are also in the same block.*

Since an FSM has only finitely many states, partition refinement will terminate. The output is the finest valid partition which is acceptable and stable. For a more formal treatment on partition refinement we refer to [65].

2.2.2 Splitting Trees and Refinable Partitions

Both types of splits described above can be used to construct a separating sequence for the states that are split. In a split w.r.t. the output after a, this sequence is simply a. In a split w.r.t. the state after a, the sequence starts with an a and continues with the separating sequence for states in $\delta(B, a)$. In order to systematically keep track of this information, we maintain a *splitting tree.* The splitting tree was introduced by Lee and Yannakakis [92] as a data structure for maintaining the operational history of a partition refinement algorithm.

Definition 2.6. *A* splitting tree *for M is a rooted tree T with a finite set of nodes with the following properties:*

- *Each node u in T is labelled by a subset of S, denoted $l(u)$.*
- *The root is labelled by S.*
- *For each inner node u, $l(u)$ is partitioned by the labels of its children.*
- *Each inner node u is associated with a sequence $\sigma(u)$ that separates states contained in different children of u.*

We use $C(u)$ to denote the set of children of a node u. The *lowest common ancestor* (lca) for a set $S' \subseteq S$ is the node u such that $S' \subseteq l(u)$ and $S' \not\subseteq l(v)$ for all $v \in C(u)$ and is denoted by $\text{lca}(S')$. For a pair of states s and t we use the shorthand $\text{lca}(s, t)$ for $\text{lca}(\{s, t\})$.

The labels $l(u)$ can be stored as a *refinable partition* data structure [141]. This is an array containing a permutation of the states, ordered so that states in the same block are adjacent. The label $l(u)$ of a node then can be indicated by a slice of this array. If node u is split, some states in the *slice* $l(u)$ may be moved to create the labels of its children, but this will not change the *set* $l(u)$.

A splitting tree T can be used to record the history of a partition refinement algorithm because at any time the leaves of T define a partition on S, denoted $P(T)$. We say a splitting tree T is valid (resp. acceptable, stable, complete) if $P(T)$ is as such. A leaf can be expanded in one of two ways, corresponding to the two ways a block can be split. Given a leaf u and its block $B = l(u)$ we define the following two splits:

split-output. Suppose there is an input a such that B can be split w.r.t output after a. Then we set $\sigma(u) = a$, and we create a node for each subset of B that produces the same output x on a. These nodes are set to be children of u.

split-state. Suppose there is an input a such that B can be split w.r.t. the state after a. Then instead of splitting B as described before, we proceed as follows. First, we locate the node $v = \text{lca}(\delta(B, a))$. Since v cannot be a leaf, it has at least two children whose labels contain elements of $\delta(B, a)$. We can use this information to expand the tree as follows. For each node w in $C(v)$ we create a child of u labelled $\{s \in B | \delta(s, a) \in l(w)\}$ if the label contains at least one state. Finally, we set $\sigma(u) = a\sigma(v)$.

A straight-forward adaptation of partition refinement for constructing a stable splitting tree for M is shown in Algorithm 1. The termination and the correctness of the algorithm outlined in Section 2.2.1 are preserved. It follows directly that states are equivalent if and only if they are in the same label of a leaf node.

Algorithm 1: Constructing a stable splitting tree

```
Input: A FSM $M$
Result: A valid and stable splitting tree $T$
initialize $T$ to be a tree with a single node labeled $S$
repeat
    find $a \in I, B \in P(T)$ such that we can split $B$ w.r.t. output $\lambda(\cdot, a)$
    expand the $u \in T$ with $l(u) = B$ as described in (split-output)
until $P(T)$ is acceptable
repeat
    find $a \in I, B \in P(T)$ such that we can split $B$ w.r.t. state $\delta(\cdot, a)$
    expand the $u \in T$ with $l(u) = B$ as described in (split-state)
until $P(T)$ is stable
```

Example 2.1. *Figure 2.1 shows a FSM and a complete splitting tree for it. This tree is constructed by Algorithm 1 as follows. First, the root node is labelled by $\{s_0, \dots, s_5\}$. The even and uneven states produce different*

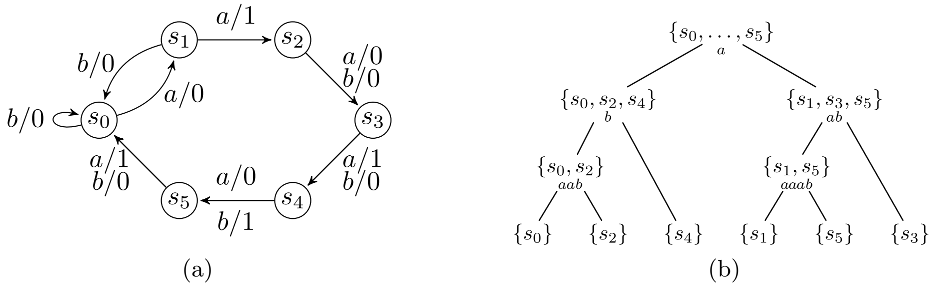


Figure 2.1: A FSM (a) and a complete splitting tree for it (b)

outputs after a, hence the root node is split. Then we note that s_4 produces a different output after b than s_0 and s_2, so $\{s_0, s_2, s_4\}$ is split as well. At this point T is acceptable: no more leaves can be split w.r.t. output. Now, the states $\delta(\{s_1, s_3, s_5\}, a)$ are contained in different leaves of T. Therefore, $\{s_1, s_3, s_5\}$ is split into $\{s_1, s_5\}$ and $\{s_3\}$ and associated with sequence ab. At this point, $\delta(\{s_0, s_2\}, a)$ contains states that are in both children of $\{s_1, s_3, s_5\}$, so $\{s_0, s_2\}$ is split and the associated sequence is aab. We continue until T is complete.

2.3 Minimal Separating Sequences

In Section 2.2.2 we have described an algorithm for constructing a complete splitting tree. This algorithm is non-deterministic, as there is no prescribed order on the splits. In this section we order them to obtain minimal separating sequences.

Let u be a non-root inner node in a splitting tree, then the sequence $\sigma(u)$ can also be used to split the parent of u. This allows us to construct splitting trees where children will never have shorter sequences than their parents, as we can always split with those sequences first. Trees obtained in this way are guaranteed to be *layered*, which means that for all nodes u and all $u' \in C(u)$, $|\sigma(u)| \leq |\sigma(u')|$. Each layer consists of nodes for which the associated separating sequences have the same length.

Our approach for constructing minimal sequences is to ensure that each layer is as large as possible before continuing to the next one. This idea is expressed formally by the following definitions.

Definition 2.7. *A splitting tree T is* k-stable *if for all states s and t in the same leaf we have $\lambda(s, x) = \lambda(t, x)$ for all $x \in I^{\leq k}$.*

Definition 2.8. *A splitting tree T is* minimal *if for all states s and t in different leaves $\lambda(s, x) \neq \lambda(t, x)$ implies $|x| \geq |\sigma(\mathrm{lca}(s, t))|$ for all $x \in I^*$.*

Minimality of a splitting tree can be used to obtain minimal separating sequences for pairs of states. If the tree is in addition stable, we obtain minimal separating sequences for all inequivalent pairs of states. Note that if a minimal splitting tree is $(n-1)$-stable (n is the number of states of M), then it is stable (Definition 2.5). This follows from the well-known fact that $n - 1$ is an upper bound for the length of a minimal separating sequence [104].

Algorithm 2 ensures a stable and minimal splitting tree. The first repeat-loop is the same as before (in Algorithm 1). Clearly, we obtain a 2-stable and minimal splitting tree here. It remains to show that we can extend this to a stable and minimal splitting tree. Algorithm 3 will perform precisely one such step towards stability, while maintaining minimality. Termination follows from the same reason as for Algorithm 1. Correctness for this algorithm is shown by the following key lemma. We will denote the input tree by T and the tree after performing Algorithm 3 by T'. Observe that T is an initial segment of T'.

Lemma 2.1. *Algorithm 3 ensures a $(k + 1)$-stable minimal splitting tree.*

Proof. Let us proof stability. Let s and t be in the same leaf of T' and let $x \in I^*$ be such that $\lambda(s, x) \neq \lambda(t, x)$. We show that $|x| > k + 1$.

Suppose for the sake of contradiction that $|x| \leq k + 1$. Let u be the leaf containing s and t and write $x = ax'$. We see that $\delta(s, a)$ and $\delta(t, a)$ are separated by k-stability of T. So the node $v = \mathrm{lca}(\delta(l(u), a))$ has children and an associated sequence $\sigma(v)$. There are two cases:

- $|\sigma(v)| < k$, then $a\sigma(v)$ separates s and t and is of length $\leq k$. This case contradicts the k-stability of T.

- $|\sigma(v)| = k$, then the loop in Alg. 3 will consider this case and split. Note that this may not split s and t (it may occur that $a\sigma(v)$ splits different elements in $l(u)$). We can repeat the above argument inductively for the newly created leaf containing s and t. By finiteness of $l(u)$, the induction will stop and, in the end, s and t are split.

Both cases end in contradiction, so we conclude that $|x| > k + 1$.

Let us now prove minimality. It suffices to consider only newly split states in T'. Let s and t be two states with $|\sigma(\text{lca}(s, t))| = k + 1$. Let $x \in I^*$ be a sequence such that $\lambda(s, x) \neq \lambda(t, x)$. We need to show that $|x| \geq k + 1$. Since $x \neq \epsilon$ we can write $x = ax'$ and consider the states $s' = \delta(s, a)$ and $t' = \delta(t, a)$ which are separated by x'. Two things can happen:

- The states s' and t' are in the same leaf in T. Then by k-stability of T we get $\lambda(s', y) = \lambda(t', y)$ for all $y \in I^{\leq k}$. So $|x'| > k$.
- The states s' and t' are in different leaves in T and let $u = \text{lca}(s', t')$. Then $a\sigma(u)$ separates s and t. Since s and t are in the same leaf in T we get $|a\sigma(u)| \geq k + 1$ by k-stability. This means that $|\sigma(u)| \geq k$ and by minimality of T we get $|x'| \geq k$.

In both cases we have shown that $|x| \geq k + 1$ as required. □

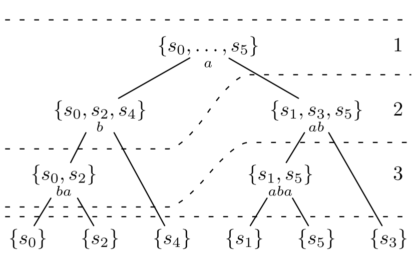
Algorithm 2: Constructing a stable and minimal splitting tree

```
**Input:** A FSM $M$ with $n$ states
**Result:** A stable, minimal splitting tree $T$
initialize $T$ to be a tree with a single node labeled $S$
**repeat**
    find $a \in I, B \in P(T)$ such that we can split $B$ w.r.t. output $\lambda(\cdot, a)$
    expand the $u \in T$ with $l(u) = B$ as described in (split-output)
**until** $P(T)$ *is acceptable*
**for** $k = 1$ *to* $n - 1$ **do**
    perform Algorithm 3 or Algorithm 4 on $T$ for $k$
```

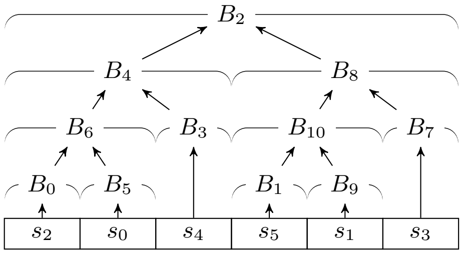
Example 2.2. *Figure 2.2a shows a stable and minimal splitting tree T for the machine in Figure 2.1a. This tree is constructed by Algorithm 2 as follows. It executes the same as Algorithm 1 until we consider the node labeled $\{s_0, s_2\}$. At this point $k = 1$. We observe that the sequence of*

Algorithm 3: A step towards the stability of a splitting tree

```
Input: a $k$-stable and minimal splitting tree $T$
Result: $T$ is a $(k+1)$-stable, minimal splitting tree
forall leaves $u \in T$ and all inputs $a$ do
    locate $v = \mathrm{lca}(\delta(l(u), a))$
    if $v$ is an inner node and $|\sigma(v)| = k$ then
        expand $u$ as described in (split-state) (which generates new
         leaves)
```



(a)



(b)

Figure 2.2: A complete and minimal splitting tree for the FSM in Figure 2.1a (a) and its internal refinable partition data structure (b)

lca($\delta(\{s_0, s_2\}, a)$) *has length* 2*, which is too long, so we continue with the next input. We find that we can indeed split w.r.t. the state after b, so the associated sequence is ba. Continuing, we obtain the same partition as before, but with smaller witnesses.*

The internal data structure (a refinable partition) is shown in Figure 2.2b: the array with the permutation of the states is at the bottom, and every block includes an indication of the slice containing its label and a pointer to its parent (as our final algorithm needs to find the parent block, but never the child blocks).

2.4 Optimizing the Algorithm

In this section, we present an improvement on Algorithm 3 that uses two ideas described by Hopcroft in his seminal paper on minimizing finite automata [72]: *using the inverse transition set*, and *processing the smaller half*. The algorithm that we present is a drop-in replacement, so that Algorithm 2 stays the same except for some bookkeeping. This way, we can establish correctness of the new algorithms more easily. The variant presented in this section reduces the amount of redundant computations that were made in Algorithm 3.

Using Hopcroft's first idea, we turn our algorithm upside down: instead of searching for the lca for each leaf, we search for the leaves u for which $l(u) \subseteq \delta^{-1}(l(v), a)$, for each potential lca v and input a. To keep the order of splits as before, we define *k-candidates*.

Definition 2.9. *A* k-candidate *is a node* v *with* $|\sigma(v)| = k$.

A k-candidate v and an input a can be used to split a leaf u if $v = \mathrm{lca}(\delta(l(u), a))$, because in this case there are at least two states s, t in $l(u)$ such that $\delta(s, a)$ and $\delta(t, a)$ are in labels of different nodes in $C(v)$. Refining u this way is called *splitting u with respect to* (v, a). The set $C(u)$ is constructed according to (split-state), where each child $w \in C(v)$ defines a child u_w of u with states

$$\begin{aligned} l(u_w) &= \{s \in l(u) \mid \delta(s, a) \in l(w)\} \\ &= l(u) \cap \delta^{-1}(l(w), a) \end{aligned} \tag{2.1}$$

In order to perform the same splits in each layer as before, we maintain a list L_k of k-candidates. We keep the list in order of the construction of nodes, because when we split w.r.t. a child of a node u before we split w.r.t. u, the result is not well-defined. Indeed, the order on L_k is the same as the order used by Algorithm 2. So far, the improved algorithm still would have time complexity $\mathcal{O}(mn)$.

To reduce the complexity we have to use Hopcroft's second idea of *processing the smaller half*. The key idea is that, when we fix a k-candidate v, all leaves are split with respect to (v, a) simultaneously. Instead of iterating over of all leaves to refine them, we iterate over $s \in \delta^{-1}(l(w), a)$ for all w in $C(v)$ and look up in which leaf it is contained to move s out of

Algorithm 4: A better step towards the stability of a splitting tree

```
   Input: a $k$-stable and minimal splitting tree $T$, and a list $L_k$
   Result: $T$ is a $(k+1)$-stable and minimal splitting tree, and a list
           $L_{k+1}$
1  $L_{k+1} \leftarrow \emptyset$
2  forall $k$-candidates $v$ in $L_k$ in order do
3      let $w'$ be a node in $C(v)$ such that $|l(w')| \geq |l(w)|$ for all nodes $w$
        in $C(v)$
4      forall inputs $a$ in $I$ do
5          forall nodes $w$ in $C(v) \setminus w'$ do
6              forall states $s$ in $\delta^{-1}(l(w), a)$ do
7                  locate leaf $u$ such that $s \in l(u)$
8                  if $C'(u)$ does not contain node $u_w$ then
9                      add a new node $u_w$ to $C'(u)$
10                 move $s$ from $l(u)$ to $l(u_w)$
11         foreach leaf $u$ with $C'(u) \neq \emptyset$ do
12             if $|l(u)| = 0$ then
13                 if $|C'(u)| = 1$ then
14                     recover $u$ by moving its elements back and clear
                        $C'(u)$
15                     continue with the next leaf
16                 set $p = u$ and $C(u) = C'(u)$
17             else
18                 construct a new node $p$ and set $C(p) = C'(u) \cup \{u\}$
19                 insert $p$ in the tree in the place where $u$ was
20             set $\sigma(p) = a\sigma(v)$
21             append $p$ to $L_{k+1}$ and clear $C'(u)$
```

it. From Lemma 8 in [84] it follows that we can skip one of the children of v. This lowers the time complexity to $\mathcal{O}(m \log n)$. In order to move s out of its leaf, each leaf u is associated with a set of temporary children $C'(u)$ that is initially empty, and will be finalized after iterating over all s and w.

In Algorithm 4 we use the ideas described above. For each k-candidate v and input a, we consider all children w of v, except for the largest one (in case of multiple largest children, we skip one of these arbitrarily). For each state $s \in \delta^{-1}(l(w), a)$ we consider the leaf u containing it. If this leaf does not have an associated temporary child for w we create such a child (line 9), if this child exists we move s into that child (line 10).

Once we have done the simultaneous splitting for the candidate v and input a, we finalize the temporary children. This is done at lines 11–21. If there is only one temporary child with all the states, no split has been made and we recover this node (line 14). In the other case we make the temporary children permanent.

The states remaining in u are those for which $\delta(s, a)$ is in the child of v that we have skipped; therefore we will call it the *implicit child*. We should not touch these states to keep the theoretical time bound. Therefore, we construct a new parent node p that will "adopt" the children in $C'(u)$ together with u (line 16).

We will now explain why considering all but the largest children of a node lowers the algorithm's time complexity. Let T be a splitting tree in which we color all children of each node blue, except for the largest one. Then:

Lemma 2.2. *A state s is in at most $(\log_2 n) - 1$ labels of blue nodes.*

Proof. Observe that every blue node u has a sibling u' such that $|l(u')| \geq |l(u)|$. So the parent $p(u)$ has at least $2|l(u)|$ states in its label, and the largest blue node has at most $n/2$ states.

Suppose a state s is contained in m blue nodes. When we walk up the tree starting at the leaf containing s, we will visit these m blue nodes. With each visit we can double the lower bound of the number of states. Hence $n/2 \geq 2^m$ and $m \leq (\log_2 n) - 1$. □

Corollary 2.1. *A state s is in at most $\log_2 n$ sets $\delta^{-1}(l(u), a)$, where u is a blue node and a is an input in I.*

If we now quantify over all transitions, we immediately get the following result. We note that the number of blue nodes is at most $n-1$, but since this fact is not used, we leave this to the reader.

Corollary 2.2. *Let $\mathcal{B}$ denote the set of blue nodes and define*

$$\mathcal{X} = \{(b, a, s) \mid b \in \mathcal{B}, a \in I, s \in \delta^{-1}(l(b), a)\}.$$

Then $\mathcal{X}$ has at most $m \log_2 n$ elements.

The important observation is that when using Algorithm 4 we iterate in total over every element in $\mathcal{X}$ at most once.

Theorem 2.1. *Algorithm 2 using Algorithm 4 runs in $\mathcal{O}(m \log n)$ time.*

Proof. We prove that bookkeeping does not increase time complexity by discussing the implementation.

Inverse transition. δ^{-1} can be constructed as a preprocessing step in $\mathcal{O}(m)$.

State sorting. As described in Section 2.2.2, we maintain a refinable partition data structure. Each time new pair of a k-candidate v and input a is considered, leaves are split by performing a bucket sort.

First, buckets are created for each node in $w \in C(v) \setminus w'$ and each leaf u that contains one or more elements from $\delta^{-1}(l(w), a)$, where w' is a largest child of v. The buckets are filled by iterating over the states in $\delta^{-1}(l(w), a)$ for all w. Then, a pivot is set for each leaf u such that exactly the states that have been placed in a bucket can be moved right of the pivot (and untouched states in $\delta^{-1}(l(w'), a)$ end up left of the pivot). For each leaf u, we iterate over the states in its buckets and the corresponding indices right of its pivot, and we swap the current state with the one that is at the current index. For each bucket a new leaf node is created. The refinable partition is updated such that the current state points to the most recently created leaf.

This way, we assure constant time lookup of the leaf for a state, and we can update the array in constant time when we move elements out of a leaf.

Largest child. For finding the largest child, we maintain counts for the temporary children and a current biggest one. On finalizing the temporary children we store (a reference to) the biggest child in the node, so that we can skip this node later in the algorithm.

Storing sequences. The operation on line 20 is done in constant time by using a linked list.

□

2.5 Application in Conformance Testing

A splitting tree can be used to extract relevant information for two classical test generation methods: a *characterization set* for the W-method and a *separating family* for the HSI-method. For an introduction and comparison of FSM-based test generation methods we refer to [47].

Definition 2.10. *A set $W \subset I^*$ is called a* characterization set *if for every pair of inequivalent states s, t there is a sequence $w \in W$ such that $\lambda(s, w) \neq \lambda(t, w)$.*

Lemma 2.3. *Let T be a complete splitting tree, then $\{\sigma(u) | u \in T\}$ is a characterization set.*

Proof. Let $W = \{\sigma(u) | u \in T\}$. Let $s, t \in S$ be inequivalent states, then by completeness s and t are contained in different leaves of T. Hence $u = lca(s, t)$ exists and $\sigma(u)$ separates s and t. Furthermore $\sigma(u) \in W$. This shows that W is a characterisation set. □

Lemma 2.4. *A characterization set with minimal length sequences can be constructed in time $\mathcal{O}(m \log n)$.*

Proof. By Lemma 2.3 the sequences associated with the inner nodes of a splitting tree form a characterization set. By Theorem 2.1, such a tree can be constructed in time $\mathcal{O}(m \log n)$. Traversing the tree to obtain the characterization set is linear in the number of nodes (and hence linear in the number of states). □

Definition 2.11. *A collection of sets $\{H_s\}_{s \in S}$ is called a* separating family *if for every pair of inequivalent states s, t there is a sequence h such that $\lambda(s, h) \neq \lambda(t, h)$ and h is a prefix of some $h_s \in H_s$ and some $h_t \in H_t$.*

Lemma 2.5. *Let T be a complete splitting tree, the sets $\{\sigma(u) | s \in l(u), u \in T\}_{s \in S}$ form a separating family.*

Proof. Let $H_s = \{\sigma(u) | s \in l(u)\}$. Let $s, t \in S$ be inequivalent states, then by completeness s and t are contained in different leaves of T. Hence $u = lca(s, t)$ exists. Since both s and t are contained in $l(u)$, the separating sequence $\sigma(u)$ is contained in both sets H_s and H_t. Therefore, it is a (trivial) prefix of some word $h_s \in H_s$ and some $h_t \in H_t$. Hence $\{H_s\}_{s \in S}$ is a separating family. □

Lemma 2.6. *A separating family with minimal length sequences can be constructed in time $\mathcal{O}(m \log n + n^2)$.*

Proof. The separating family can be constructed from the splitting tree by collecting all sequences of all parents of a state (by Lemma 2.5). Since we have to do this for every state, this takes $\mathcal{O}(n^2)$ time. □

For test generation one moreover needs a transition cover. This can be constructed in linear time with a breadth first search. We conclude that we can construct all necessary information for the W-method in time $\mathcal{O}(m \log n)$ as opposed to the $\mathcal{O}(mn)$ algorithm used in [47]. Furthermore, we conclude that we can construct all the necessary information for the HSI-method in time $\mathcal{O}(m \log n + n^2)$, improving on the reported bound $\mathcal{O}(mn^3)$ in [70]. The original HSI-method was formulated differently and might generate smaller sets. We conjecture that our separating family has the same size if we furthermore remove redundant prefixes. This can be done in $\mathcal{O}(n^2)$ time using a trie data structure.

2.6 Experimental Results

We have implemented Algorithms 3 and 4 in Go, and we have compared their running time on two sets of FSMs.[1] The first set is from [127], where

[1] Available at `https://gitlab.science.ru.nl/rick/partition/`.

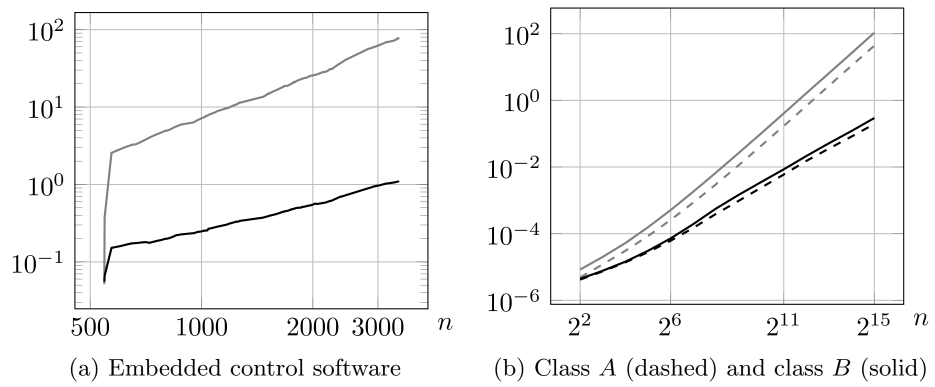


Figure 2.3: Running time in seconds of Algorithm 3 (gray) and Algorithm 4 (black)

FSMs for embedded control software were automatically constructed. These FSMs are of increasing size, varying from 546 to 3 410 states, with 78 inputs and up to 151 outputs. The second set is inferred from [72], where two classes of finite automata, A and B, are described that serve as a worst case for Algorithms 3 and 4 respectively. The FSMs that we have constructed for these automata have 1 input, 2 outputs, and 2^2 – 2^{15} states. The running times in seconds on an Intel Core i5-2500 are plotted in Figure 2.3. We note that different slopes imply different complexity classes, since both axes have a logarithmic scale.

2.7 Conclusion

In this chapter we have described an efficient algorithm for constructing a set of minimal-length sequences that pairwise distinguish all states of a finite state machine. By extending Hopcroft's minimization algorithm, we are able to construct such sequences in $\mathcal{O}(m \log n)$ for a machine with m transitions and n states. This improves on the traditional $\mathcal{O}(mn)$ method that is based on the classic algorithm by Moore. As an upshot, the sequences obtained form a characterization set and a separating family, which play a crucial in conformance testing.

Two key observations were required for a correct adaptation of Hopcroft's algorithm. First, it is required to perform splits in order of the length of their associated sequences. This guarantees minimality of the obtained separating sequences. Second, it is required to consider nodes as a candidate before any one of its children are considered as a candidate. This order follows naturally from the construction of a splitting tree.

Experimental results show that our algorithm outperforms the classic approach for both worst-case finite state machines and models of embedded control software. Applications of minimal separating sequences such as the ones occurring in [47, 127] therefore show that our algorithm is useful in practice.

Chapter 3

Bigger is Not Always Better

Rick Smetsers, Michele Volpato, Frits Vaandrager, and Sicco Verwer

Abstract

In Angluin's L^* algorithm a learner iteratively constructs hypotheses in order to learn a regular language. Each hypothesis is consistent with a larger set of observations and is described by a bigger model. From a behavioral perspective, however, a hypothesis is not always better than the previous one, in the sense that the minimal length of a counterexample that distinguishes a hypothesis from the target language may decrease. We present a simple modification of the L^* algorithm that ensures that for subsequent hypotheses the minimal length of a counterexample never decreases, which implies that the distance to the target language never increases in a corresponding ultrametric. Preliminary experimental evidence suggests that our algorithm speeds up learning in practical applications by reducing the number of equivalence queries.

3.1 Introduction

Automata learning techniques have become increasingly important for their applications to a wide variety of software engineering problems, especially in the analysis and testing of complex systems. Recently, they have been successfully applied for security protocol testing [125], for the analysis of botnet command and control protocols [40], in regression testing of telecommunication protocols [77], and in conformance testing of communication protocols [6].

Automata learning aims to identify an unknown target language from examples of its members and nonmembers [63]. In *active* automata learning, introduced in the seminal work by Angluin [11], a *learner* identifies the language with the help of an *oracle* (in contrast to *passive* learning, where the learner is provided with data). Angluin's L^* algorithm is characterized by the iterative alternation between two phases. In the first phase, the learner poses *membership queries* to construct a hypothesis. In the second phase it asks an *equivalence query* to determine if the hypothesis correctly describes the language. The oracle either signals success (if the hypothesis correctly describes the language) or provides a counterexample that distinguishes the hypothesis and the language. The algorithm iterates in this way until it finds a hypothesis that correctly describes the target language. In the learning process, each successive hypothesis is described by a bigger model.

In this chapter, we show that a bigger model is not always better. Different notions of quality exist for a hypothesis and we argue that a valid metric for quality should be based on its *behaviour*, i.e. the strings in its language. In systems engineering, a potential bug in the far-away future is less troubling than a potential bug today [9]. Based on this observation, we study a well-known metric based on minimal-length counterexamples and we show that the quality of successive hypotheses may decrease in such a setting. To correct for this, we propose a simple modification to L^* that finds a counterexample at the cost of a membership query if this is the case. As a result, we make sure that each hypothesis is at least as good as the previous one, and we possibly decrease the number of equivalence queries required in the learning process. We give preliminary experimental evidence that in a realistic setting our modification speeds up learning, because in practice, equivalence queries are typically expensive to answer [126]. In a case study, we show that our modification helps in learning a piece of industrial control software used in Océ printers.

Recently, the L^* algorithm has been adapted for learning the behaviour of reactive systems [131, 124]. In practice, when learning such systems, it is impossible to exhaustively search for counterexamples; we have to stop learning at some point. As a result, the oracle is not always perfect and it is possible that the final hypothesis is incorrect. Contrary to the original L^* algorithm, our modification guarantees that in such a case the final hypothesis will behave correctly for at least as long as all previous hypotheses.

Related work. Improvements of L^* have been investigated before; we refer to Balcázar et al. for an overview of early work [15]. Optimisations for large alphabets are discussed in [81], and practical optimisations that improve both the number of queries and time required to answer them are given in [16] and [80]. Distance metrics for automata have been studied extensively before, but the majority of this work has been done in a setting where statistical information is available [71]. To the best of our knowledge, no earlier work has compared successive hypotheses produced by L^* from the perspective of their counterexamples. Length-bounded counterexamples have been used in [79], and minimal-length counterexamples in [21] and [78]. These are, however, strong assumptions. Instead, our modification works in a standard L^* setting.

Outline. We recall regular languages, Angluin's L^* algorithm and metric spaces in Section 3.2. Then, in Section 3.3, we define a metric for comparing languages, and we show that L^* may construct hypotheses that are worse than previous ones according to this metric. Section 3.4 provides an algorithm that solves this problem and some preliminary experimental results. We conclude our work in Section 3.6.

3.2 Preliminaries

Definitions. Let Σ be a finite *alphabet* of *symbols*. A *string* over Σ is a finite sequence of symbols. We denote with Σ^* the set of all strings over Σ. The empty string is denoted by ϵ. A *language* is a subset of Σ^*. We denote

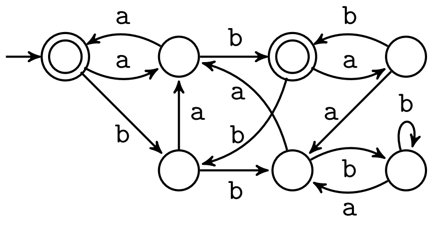


Figure 3.1: A canonical DFA over the alphabet $\{\mathtt{a}, \mathtt{b}\}$. Elements of Q are represented by nodes, and elements of F by double circle nodes. The initial state is indicated by the non-labelled arrow. An edge between states p and q, labeled with an element a of the alphabet, is present if and only if $\delta(p, a) = q$.

$\mathcal{L}_\Sigma$ the set of all languages over the alphabet Σ, and we shorten it to $\mathcal{L}$ if Σ is clear or not significant in the context.

A *regular* language is any language that is accepted by some *deterministic finite automaton* (DFA), which is a tuple $A = \langle \Sigma, Q, q_0, F, \delta \rangle$, where Σ is a finite alphabet, Q is a finite set of *states*, $q_0 \in Q$ is the *initial* state, $F \subseteq Q$ is the set of *accepting* states, and $\delta : Q \times \Sigma \to Q$ is the *transition function* between states. We extend δ to $Q \times \Sigma^* \to Q$ in the usual way. The language accepted by A is the set of strings u such that $\delta(q_0, u) \in F$, and is denoted L_A. Two DFAs A and A' are *equivalent*, denoted $A \equiv A'$ if they accept the same language, i.e $L_A = L_{A'}$. A DFA is *canonical* if no other equivalent DFA has fewer states. In the rest of the chapter, all DFAs are considered to be canonical, unless specified otherwise. Figure 3.2 shows an example canonical DFA that we will use throughout this chapter.

We say that a string u *distinguishes* A and A' if $u \in L_A \iff u \notin L_{A'}$. Such a string is called a *distinguishing string* for L_A and $L_{A'}$. A *minimal-length distinguishing string* for two languages L and L' is defined as a string v, such that, for each string w, $|w| < |v|$ implies $w \in L \iff w \in L'$. There exist efficient algorithms for finding a minimal-length distinguishing string between two DFAs (see for example [48]).

Learning regular languages with L^*. Angluin's L^* algorithm is an efficient algorithm where a *learner* identifies an unknown regular language L

with the help of an *oracle*. The oracle answers two types of queries about the target language. In a *membership query* the learner asks if a string is in the language. After having posed a number of membership queries, the learner constructs a canonical DFA A_H that is consistent with all the replies given by the oracle so far. The language H that this DFA accepts is the learner's *hypothesis* for the target language. Depending on the context, the word "hypothesis" is either used to refer to a language H, or to the canonical DFA that accepts this language A_H. In an *equivalence query* the learner asks about the correctness of H. The oracle replies positively if $H = L$. If this is not the case, the oracle returns a distinguishing string that shows that the hypothesis is incorrect. Such a string is called a *counterexample*. An oracle that answers these two types of queries is known as a *minimally adequate teacher*.

The L^* algorithm maintains an *observation table* in which it stores the answers to all membership queries posed so far. The observation table consists of a set of *access strings* to the states of the hypothesis DFA, their *one-symbol extensions*, and a set of *distinguishing suffixes*. An observation table can be visualized as a table that has the access strings and their one-symbol extensions as its row labels and the distinguishing suffixes as its column labels. A cell has a value of 2 if and only if the concatenation of the corresponding prefix (access sequence or one-symbol extension) and (distinguishing) suffix is in the language. Otherwise, a cell has a value of 0. If all cells are filled, the table is *complete*. Example complete observation tables are shown in Table 3.2.

In order to construct a hypothesis DFA from the observation table, it needs to be *closed* and *consistent*. An observation table is *closed* if for each row labeled with a one-symbol extension there exists a row with an access string that has an identical value in every column. An observation table is *consistent* if for all rows labeled with access strings that have an identical value in every column, it holds that the rows labeled by their one-symbol extensions have an identical value in every row as well.

We provide a high-level description of L^* (Algorithm 5). For a more detailed description we refer to [71]. First, the observation table is initialized such that the set of access strings and the set of distinguishing strings both contain ϵ, and the algorithms asks membership queries for ϵ and Σ (the one-symbol extensions of ϵ). Then, the table is checked for closedness and

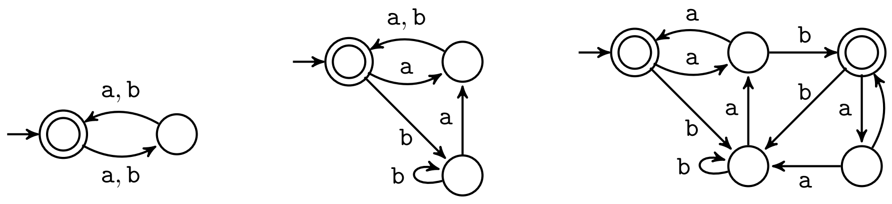
Algorithm 5: The L^* algorithm

```
Initialize observation table
Make table closed and consistent
**repeat**
    Construct hypothesis $H$
    Ask equivalence query for $H$, let $c$ be the response
    **if** $c$ *contains a counterexample* **then**
        Handle counterexample $c$
        Make the table closed and consistent
**until** $c$ *does not contain a counterexample*
```

consistency. If the table is not closed or not consistent, an element from the one-symbol extensions is added to the access strings, and its one-symbol extensions are added to the table (a new state is added). This process is repeated until the table is closed and consistent. Once the table is closed and consistent, a hypothesis DFA $A = \langle \Sigma, Q, q_0, F, \delta \rangle$ is constructed in the following way:

- Q contains exactly one state for every access sequence. This ensures that for every pair of states, the corresponding rows have different values in at least one column;
- q_0 is the state in Q for access sequence ϵ;
- F contains states in Q for access sequences that are in the language;
- The one-symbol extensions are used to define δ, where $\delta(\delta(q_0, s), a) = \delta(q_0, t)$ if and only if (one-symbol extension) $s \cdot a$ and (access string) t have identical values in every column.

The hypothesis is presented to the oracle in an equivalence query. If the oracle replies positively, the learner has successfully learned the target language and the algorithm terminates. Otherwise, the oracle provides a counterexample and the algorithm modifies the observation table. There are many different strategies for handling a counterexample (see e.g. [98, 119, 124, 131]). In this chapter we use the strategy described by Steffen et al. [131], but our contribution is independent from the one that is used.



(a) Initial hypothesis. (b) Hypothesis after handling ba. (c) Hypothesis after handling abaa.

Figure 3.2: Starting the learning process.

After handling a counterexample the table is not closed anymore, so an element from the one-symbol extensions is added to the access strings (a new state is added), and more membership queries are asked to obtain a new hypothesis. The algorithm iterates in this fashion until it produces a correct hypothesis. Each hypothesis in the learning process is described by a canonical DFA, and each successive hypothesis has more states than the previous one. Assume that the canonical DFA accepting the target language has n states, then the algorithm clearly terminates, because the number of equivalence queries is limited by n. An example run of L^* is described in Example 3.1.

Example 3.1. *We show how L^* constructs the first three hypotheses while learning the language described by Figure 3.2. We start by asking membership queries for ϵ,* `a` *and* `b` *and we store the answers in an observation table. We find that the table is not closed, so we add* `a` *to S and we ask membership queries for* `aa` *and* `ab`*. Then, we construct the table shown in Table 3.2a and the corresponding hypothesis shown in Figure 3.2a. This hypothesis is presented as an equivalence query. The oracle replies that our hypothesis evaluates* `ba` *incorrectly. We handle this counterexample in the table and after asking more membership queries, we construct the table shown in Table 3.2b and we present the hypothesis shown in Figure 3.2b as an equivalence query. This time, the oracle presents* `abaa` *as a counterexample. We use this information to construct the table shown in Table 3.2c and the hypothesis shown in Figure 3.2c.*

Table 3.1: Observation tables. Rows are labeled by access strings (top) and their one-symbol extensions (bottom), columns are labeled with distinguishing suffixes.

	ϵ
ϵ	1
a	0
b	0
aa	1
ab	1

(a) Initial observation table.

	ϵ	a
ϵ	1	0
a	0	1
b	0	0
aa	1	0
ab	1	0
ba	0	1
bb	0	0

(b) Observation table after handling ba.

	ϵ	a	aa
ϵ	1	0	1
a	0	1	0
b	0	0	1
ab	1	0	0
aba	0	0	0
aa	1	0	1
ba	0	1	0
bb	0	0	1
abb	0	0	1
abaa	0	0	1
abab	1	0	0

(c) Observation table after handing abaa.

Metric spaces. In order to reason about the quality of hypotheses produced by L^*, we need a function to compare them. A function is called a *metric*, if it satisfies the conditions in Definition 3.1.

Definition 3.1. *Let X be a set, then a function $d : X \times X \to \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers, is a* metric *on X if:*

1. $d(x, y) = 0 \iff x = y$ *(identity);*
2. $d(x, y) = d(y, x)$ *(symmetry);*
3. $d(x, y) \le d(x, z) + d(z, y)$ *(triangle inequality).*

A metric is an ultrametric *if in addition it satisfies a stronger version of triangle inequality:*

4. $d(x, y) \le \max(d(x, z), d(z, y))$ *(strong triangle inequality).*

Note that any metric is nonnegative. Given a set X and a metric d on X, the pair $\langle X, d\rangle$ is called a *metric space*. If d is an ultrametric on

X, we call $\langle X, d\rangle$ an *ultrametric space.* In metric and ultrametric spaces, the function d provides the set X with the concept of distance between its elements. Given three elements $x, y, z \in X$, we say that x is *closer* to z than y if $d(x, z) < d(y, z)$. In an ultrametric space $\langle X, d\rangle$, for any triple of elements $x, y, z \in X$, two of the distances among them are equal and the third one is equal or smaller (Lemma 3.1).

Lemma 3.1. *Let $\langle X, d\rangle$ be an ultrametric space and let x, y, z be elements of X, then:*

$$d(x, y) \neq d(y, z) \implies d(x, z) = \max\left(d(x, y), d(y, z)\right).$$

Proof. Assume $d(x, y) > d(y, z)$. Then, because of strong triangle inequality, we obtain that $d(x, z) \leq \max\left(d(x, y), d(y, z)\right) = d(x, y)$ and that $d(x, y) \leq \max\left(d(y, z), d(x, z)\right) = d(x, z)$ (because $d(y, z) < d(x, y)$). Thus $d(x, y) = d(x, z)$. $\square$

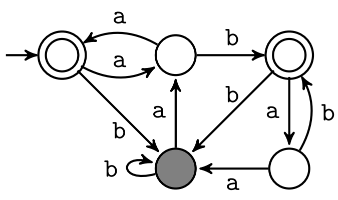
3.3 Languages in a Metric Space

In the process of learning a language, different notions of quality exist for a hypothesis. The L^* algorithm guarantees that each hypothesis explains all observations from membership queries seen so far. Moreover, each successive hypothesis is based on more observations than the previous one, because a counterexample adds new information. As a result, each hypothesis has more states than the previous one (see Figure 3.2, for example).

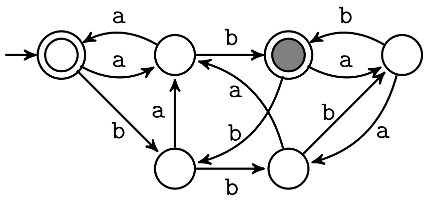
In this section, we show that a bigger model is not always better. In fact, the size difference between a hypothesis and the target DFA is not a valid metric. Let A_H and $A_{H'}$ be DFAs with m and n states respectively, and let $H, H' \in \mathcal{L}$ be the languages that they accept. Then it is possible that $m = n$ and $H \neq H'$. Hence, a function on the number of states is not a valid metric on $\mathcal{L}$, because it does not satisfy the identity axiom of Definition 3.1.

We argue that a valid metric should be based on the *behaviour* of a hypothesis, in terms of the strings in its language. Assume that, with regard to such a metric, a hypothesis is more distant to the target language than the previous one. Then, if one can detect this divergence, there must be some information that can be used to improve the hypothesis, without the need of asking an equivalence query. We recall a well-known metric for comparing two languages [14]. Intuitively, this metric is based on the minimal length of distinguishing strings: languages are more distant if they are distinguished by shorter strings. Example 3.2 shows that in L^*, the minimal length of a counterexample for hypotheses may decrease. As a result, the distance to the target language increases.

Example 3.2. *Let us continue the process for learning the language represented by Figure 3.2. The starting point is the hypothesis accepted by Figure 3.3a. In this hypothesis all strings of length 4 are evaluated correctly and we are presented* `bbbaa` *as a counterexample. While handling* `bbbaa` *something peculiar happens: our new hypothesis (Figure 3.3b) incorrectly changes behaviour for* `bbbb`*. Hence, an error has been introduced, and the quality of our hypothesis has decreased.*



(a) Third hypothesis. The string `bbbb` is evaluated correctly.



(b) Fourth hypothesis. The string `bbbb` is evaluated incorrectly.

Figure 3.3: After handling `bbbaa` (a minimal-length counterexample), a shorter string (`bbbb`) incorrectly ends in an accepting state. Grey nodes denote the state that `bbbb` ends in.

Given two different languages L_1 and L_2, let u be a minimal-length string that distinguishes them. Clearly, u distinguishes any DFA accepting L_1 from any DFA accepting L_2 and vice versa. In Definition 3.2 we present a metric to compare L_1 and L_2 based on the length of this string. Similar metrics have been used in literature, e.g. in concurrency theory [14]. Note that, if two languages are not equal, there must exist a distinguishing string. The intuition is that, the longer this string is, the closer L_1 and L_2 are.

Definition 3.2. *Let L_1 and L_2 be two languages. The ultrametric d is the function*

$$d(L_1, L_2) = \begin{cases} 0 & \text{if } L_1 = L_2 \\ 2^{-n} & \text{otherwise} \end{cases}$$

where n is the minimal length of a string that distinguishes L_1 and L_2.

Lemma 3.2. *The pair $\langle \mathcal{L}, d \rangle$ with d defined by Definition 3.2 is an ultrametric space.*

Proof. We prove that the pair $\langle \mathcal{L}, d \rangle$ with d defined by Definition 3.2 satisfies the three axioms of an ultrametric. Let L_1, L_2 and L_3 be languages in $\mathcal{L}$. Then

i) $d(L_1, L_2) = 0 \iff L_1 = L_2$: ($\Leftarrow$) by definition;
($\Rightarrow$) it must hold that $L_1 = L_2$ because $2^{-|u_{12}|}$, where u_{12} is a minimal-length string that distinguish L_2 from L_2, can never evaluate to 0;

ii) $d(L_1, L_2) = d(L_2, L_1)$: the strings that distinguish L_1 from L_2 are the same that distinguish L_2 from L_1;

iii) $d(L_1, L_2) \leq \max(d(L_2, L_3), d(L_3, L_1))$: if any two among L_1, L_2 and L_3 are equal, then it holds trivially. Otherwise, if they are all different, then let u_{12}, u_{23} and u_{31} be minimal-length strings that pairwise distinguish them. We show that $|u_{12}| \geq \min(|u_{23}|, |u_{31}|)$ (which is equivalent to $d(L_1, L_2) \leq \max(d(L_2, L_3), d(L_3, L_1))$). By contradiction, assume $|u_{12}| < \min(|u_{23}|, |u_{31}|)$. By definition of minimal-length distinguishing strings, $\forall w$ such that $(|w| < |u_{23}|) \wedge (|w| < |u_{31}|)$ it holds that $(w \in L_2 \iff w \in L_3) \wedge (w \in L_3 \iff w \in L_1)$, in particular for $w = u_{12}$. But, then, u_{12} is not a distinguishing string for L_1 and L_2 which is a contradiction.

□

In Example 3.2 we have used minimal-length counterexamples to illustrate that L^* can produce a hypothesis that is more distant to the target language than the previous one. Unfortunately, in the process of learning an unknown target language, finding a minimal-length counterexample for a hypothesis is a difficult task. We can however make use of the strong triangle inequality property of ultrametrics (see Lemma 3.1) to check a hypothesis' relative distance to the target at the cost of a single membership query. By Lemma 3.1, for the ultrametric described in Definition 3.2 this means that for any triple of elements, two of the minimal-length distinguishing strings will have equal lengths and the third will be of equal or *greater* length.

Lemma 3.3. *Let $\langle \mathcal{L}, d\rangle$ be the ultrametric space with d defined by Definition 3.2. Let L, H and H' be languages in $\mathcal{L}$ with $H \neq H'$ and let v be any minimal-length distinguishing string for H and H'. Then:*

$$d(L,H) < d(L,H') \implies (v \in H \iff v \in L)$$

Proof. Let u and u' be minimal-length strings distinguishing L from H and H', respectively. Then by Definition 3.2, $d(L,H) < d(L,H')$ implies that $|u| > |u'|$. Moreover, by Lemma 3.1 $d(L,H) < d(L,H')$ implies that $d(L,H') = d(H,H')$ which in turn implies that $|v| = |u'|$, and hence that $|u| > |v|$. It follows that $v \in H \iff v \in L$, because u is a minimal-length distinguishing string. □

Assume that H and H' are successive hypotheses for L and that v is a minimal-length string that distinguishes the two hypotheses. Then to verify that H' is at least as close as H it suffices to ask a membership query for v. If H evaluates v incorrectly, then according to Lemma 3.3, H' is at least as close as H. If H evaluates v correctly, we cannot be sure that this is the case, but we have found a new counterexample for H' without asking an equivalence query. The algorithm that we present in Section 3.4 makes use of this result to guarantee that the distance to L does not increase.

3.4 Monitoring the Quality of Hypotheses

In the previous section we have seen that L^* can change the behaviour of strings that happened to be handled correctly before. As a result, the distance of a hypothesis to the target language can increase. In this section, we propose a modification to L^* which guarantees that the distance to the target language does not increase in the ultrametric space $\langle \mathcal{L}, d\rangle$, with d defined in Definition 3.2.

Definition 3.3. *Let H and H' be successive hypotheses in process of identifying an unknown language L, and let d be the ultrametric defined in Definition 3.2, then H' is* stable *if and only if $d(H,L) \geq d(H',L)$.*

Our modification to the L^* algorithm is shown in Algorithm 6. In the algorithm we maintain a *stable* hypothesis H that is known to be the best we have seen so far according to Definition 3.3. The main idea is that a *candidate* hypothesis H' is refined until it is at least as good as the stable hypothesis before asking an equivalence query. As a result, each stable hypothesis is at least as good as the previous one.

Algorithm 6: Modified L^* algorithm

```
Initialize observation table
Make table closed and consistent
**repeat**
    Construct stable hypothesis $H$
    Ask equivalence query for $H$, let $c$ be the response
    **if** *$c$ contains a counterexample* **then**
        Handle counterexample $c$
        Make table closed and consistent
        **repeat**
            Construct candidate hypothesis $H'$
            Obtain minimal-length string $v$ that distinguishes $H$ and
              $H'$
            **if** *$v$ distinguishes $H'$ and $L$* **then**
                Handle counterexample $v$
                Make table closed and consistent
        **until** *$v$ distinguishes $H$ and $L$*
**until** *$c$ does not contain a counterexample*
```

We start by constructing the initial stable hypothesis H that we present in an equivalence query. If the oracle replies positively, the learner has successfully learned the target language and the algorithm terminates. If this is not the case we obtain a counterexample c, which we use to construct a candidate hypothesis H'. Then, we use a standard algorithm to obtain a minimal-length string v that distinguishes the stable hypothesis and the candidate hypothesis.

According to Lemma 3.3, we are sure that the candidate (H') is at least as good as the previous stable hypothesis (H) if v is evaluated correctly by the candidate H'. If, however, v is evaluated correctly by the previous stable hypothesis H (and the condition is not met), then we cannot guarantee that the candidate has improved. In this case, we have however found a counterexample (v) without asking an equivalence query. We use this counterexample to construct a new (and bigger) candidate H' (following the standard L^* procedure) and we again ask for a minimal-length string v that distinguishes the new candidate from the stable hypothesis. The algorithm iterates in this way until it finds a minimal-length distinguishing string v that is evaluated correctly by the candidate. In this case we break the loop, promote the candidate to stable and ask for the next equivalence query. The correctness and termination of our algorithm are proven by Theorem 3.1.

Theorem 3.1. *The execution of Algorithm 6 terminates, and each stable hypothesis is at least as good as the previous one.*

Proof. First, note that the inner loop handles a counterexample in the regular way. Hence, each candidate hypothesis has more states than any previous hypothesis. We show that the number of iterations in the inner loop is finite, which proves that the algorithm terminates, given the termination of the original L^* algorithm.

Let H' be the candidate hypothesis obtained from H by handling c, and let v be a minimal-length distinguishing string for H' and H. Let u and u' be (unknown) minimal-length counterexamples of H and H' respectively, and note that u can never be longer than c. Moreover, note that v can never be longer than c because, by construction, $c \in H \iff c \notin H'$ and v is a string that distinguishes H and H'.

If v distinguishes H and L, then v is a counterexample for H. Hence, by contraposition of Lemma 3.3, $d(L, H) \geq d(L, H')$. The algorithm breaks out of the inner loop and promotes H' to stable. If, instead v distinguishes H' and L, then it might be the case that $|u| > |u'|$ (and hence that $d(L, H) < d(L, H')$). Thus, given that H' evaluates v incorrectly, v can be used as a counterexample obtaining a new hypothesis H'' and a new minimal-length distinguishing string v'' for H'' and H. Note that v does not distinguish H'' and L, and hence that $v \neq v''$. Consequently, in successive loops, the algorithm never processes v again, because all hypotheses are consistent with the membership queries asked so far. Given that there are a finite number of strings that are not longer than c, the inner loop terminates. $\square$

In Example 3.3 we show that, by using Algorithm 6, each stable hypothesis is at least as good as the previous one. As a result, we reduce the number of equivalence queries required in learning the language represented by Figure 3.2.

Example 3.3. *Let us continue the process for learning the language represented by the DFA in Figure 3.2. Our starting point is the hypothesis in Figure 3.3a. When this hypothesis is presented as an equivalence query, the oracle provides* `bbbaa` *as a counterexample. By handling this counterexample, we obtain the hypothesis in Figure 3.3b. The original L^* would present this hypothesis to the oracle in an equivalence query. Instead, Algorithm 6 first verifies that the hypothesis is at least as good as the previous one. To do so, it finds a minimal-length string (*`bbbb`*) that distinguishes it from the previous hypothesis (Figures 3.3a). Then, it asks a membership query to check if the behaviour has changed. The membership query returns* `false`*, and so does the previous hypothesis. Hence, an error was introduced in the current hypothesis. Instead of presenting the hypothesis as an equivalence query, it is refined by handling* `bbbb`*.*

For the refined hypothesis, the algorithm performs another inequivalence check. This time, a minimal-length string that distinguishes the current hypothesis from the previous one is `bbbaa`*, the initial counterexample. This string is evaluated incorrectly in the previous hypothesis (and correctly in the current one). Therefore, the algorithm exits the loop and continues learning. The current hypothesis is, presented as an equivalence query, and the oracle replies affirmatively. Algorithm 6 required four equivalence queries to learn the target language, one less than the original L^* algorithm. Moreover, the algorithm guarantees that each hypothesis is at least as good as the previous one, contrary to the original L^* algorithm.*

3.5 Experimental Results

In this section, we give preliminary experimental evidence that our algorithm speeds up learning. We have implemented Algorithm 6 on top of L^* in LearnLib, a state-of-the-art tool for active automata learning [117, 102], and we have compared the number of equivalence queries that it requires. Compared to an implementation of the original L^* algorithm, Algorithm 6 requires approximately 4% less equivalence queries. The experiments were conducted on randomly generated DFAs with a varying number of states (100–2000) and alphabet (2–20).

To show that the contributions of this chapter are suited for practical learning problems as well, we have applied them in a more realistic setting. Recently, automata learning techniques have become increasingly important for the construction of models for software components. Smeenk et al. [126, 127] used the L^* algorithm to learn the behaviour of the Engine Status Manager (ESM), a piece of industrial software that controls the transition from one status to another in Océ printers[1]. Learning the behaviour of this software is hard because of the many details involved. A key challenge that the authors faced was the task of searching for a counterexample: more than 263 million sequences of input actions were not enough to fully learn the behaviour of the system. As a result, the learning process was terminated before the correct hypothesis was found. In total, the time required for learning exceeded 19 hours, of which over 7 hours were spent on searching for counterexamples. Altogether, 131 hypotheses were generated. The partially correct final hypothesis had 3,326 states.

Using the distance metric described in Section 3.3 we were able to verify that the partially correct final hypothesis was the best one seen so far. However, by comparing intermediate hypotheses (Algorithm 6), we have found a counterexample for four hypotheses without having to search for them using an equivalence query. In these cases, a minimal-length distinguishing string for two successive hypotheses had incorrectly changed its behaviour. With the use of our algorithm, we would have been able to detect these mistakes. As a result, it is highly likely that our algorithm would have reduced the time required to learn the final hypothesis. This case study shows the implications of our contributions in practice: behaviour-based metrics can provide useful information about hypotheses in the learning process. Using this information, we reduce the number of equivalence queries required.

[1]A detailed description of the case study, with models and statistics, is available at `http://www.mbsd.cs.ru.nl/publications/papers/fvaan/ESM/`.

3.6 Conclusion

Bigger hypotheses in active automata learning are not always better. In this chapter, we have shown that different notions of quality exist for a hypothesis. We have argued that a valid metric should be based on the *behaviour* of a hypothesis. To the best of our knowledge, our work is the first to address the quality of hypotheses in active learning from such a solid, theoretical perspective.

Using a well-known metric based on minimal-length counterexamples, we have shown that the quality of successive hypotheses produced by L^* may decrease. To correct this, we have proposed a simple modification to L^* that makes sure that each hypothesis is at least as good as the previous one.

Experiments and a case study have provided preliminary evidence that our contributions are effective in practice. Moreover, they have shown that behaviour-based metrics can provide useful information about the learning process, that can be used to reduce the number of equivalence queries required in active learning.

The results of this chapter may provide insights in the problem of finding counterexamples for an hypothesis. In a realistic setting, where the help of an oracle is unavailable, we have to search for counterexamples by posing membership queries. In our experiments, we have shown that a minimal-length distinguishing string for successive hypotheses has a relatively high chance to be a counterexample. In future work, we wish to investigate if other distinguishing strings are good candidates as well. A search strategy based on these strings might find a counterexample more quickly.

Chapter 4

Enhancing Automata Learning by Log-Based Metrics

Petra van den Bos, Rick Smetsers, and Frits Vaandrager

Abstract

We study a general class of distance metrics for deterministic Mealy machines. The metrics are induced by weight functions that specify the relative importance of input sequences. By choosing an appropriate weight function we may fine-tune a metric so that it captures some intuitive notion of quality. In particular, we present a metric that is based on the minimal number of inputs that must be provided to obtain a counterexample, starting from states that can be reached by a given set of logs. For any weight function, we may boost the performance of existing model learning algorithms by introducing an extra component, which we call the *Comparator*. Preliminary experiments show that the Comparator yields a significant reduction of the number of inputs required to learn correct models. Moreover, by generalising a result of Chapter 3, we show that the quality of hypotheses generated by the Comparator never decreases.

4.1 Introduction

In the platonic boolean world view of classical computer science, which goes back to McCarthy, Hoare, Dijkstra and others, programs can only be correct or incorrect. Henzinger [67] argues that this boolean classification falls short of the practical need to assess the behaviour of software in a more nuanced fashion against multiple criteria. He proposes to introduce quantitative fitness measures for programs, in order to measure properties such as functional correctness, performance and robustness. This chapter introduces such quantitative fitness measures in the context of black-box testing, an area in which, as famously observed by Dijkstra [46], it is impossible to establish correctness of implementations.

The scenario that we consider in this chapter starts from some legacy software component. Being able to retrieve models of such a component is potentially very useful. For instance, if the software is changed or enriched with new functionality, one may use a learned model for regression testing. Also, if the source code is hard to read and poorly documented, one may use a model of the software for model-based testing of a new implementation, or even for generating an implementation on a new platform automatically.

The construction of models from observations of component behaviour can be performed using model learning (e.g. regular inference) techniques [71]. One such technique is *active learning* [11, 131]. In active learning, a so-called Learner interacts with a System Under Learning (SUL), which is a black-box reactive system the Learner can provide inputs to and observe outputs from. By interacting with the SUL, the Learner infers a *hypothesis*, a state machine model that intends to describe the behaviour of the SUL. In order to find out whether a hypothesis is correct, we will typically use some conformance testing method. If the SUL passes the test, then the model is deemed correct. If the outputs of the SUL and the model differ, the test constitutes a counterexample, which may then be used by the Learner to construct an improved hypothesis.

Active learning has been successfully applied to learn models of (and find mistakes in) implementations of major protocols such as TCP [55, 54] and TLS [122]. We have also used the approach to learn models of embedded control software at Océ [127] and to support refactoring of software at Philips HealthTech [123]. A key issue in black-box model learning, however, is assessing the correctness of the learned models. Since testing may fail to find a counterexample for a model, we can never be sure that a learned model is correct. Hence there is an urgent need for appropriate quantitative fitness measures.

Given a correct model S of the behaviour of the SUL, and a hypothesis model H, we are interested in distance metrics d that satisfy the following three criteria:

1. $d(H, S) = 0$ iff H and S have the same behaviour.
2. For any $\varepsilon > 0$, there exists a finite test suite T_ε such that if H and S behave the same for all tests in T_ε, it follows that $d(H, S) < \varepsilon$.
3. Metric d captures some intuitive notion of quality: the smaller $d(H, S)$, the better the quality of hypothesis H.

The first criterion is an obvious sanity property that any metric should satisfy. The second criterion says that, even though we can never exclude that H and S behave differently, we may, for any $\varepsilon > 0$, come up with a finite test suite T_ε to check whether $d(H, S) < \varepsilon$. By running all the tests in T_ε we either establish that H is a ε-approximation of S, or we find a counterexample that we can use to further improve our hypothesis model H. The third criterion is somewhat vague, but nevertheless extremely important. In practice, engineers will only be willing to invest further in testing if this leads to a quantifiable increase of demonstrated quality. They usually find it difficult to formalise their intuitive concept of quality, but typically require that a refactored implementation of a legacy component behaves the same for a set of common input sequences that have been recorded in log files, or specified as part of a regression test suite.

In this chapter, we introduce a new, general class of metrics for deterministic Mealy machines that satisfy criteria (1) and (2). Our metrics are induced by weight functions that specify the relative importance of input sequences. By choosing an appropriate weight function we may fine-tune our metric so that it also meets criterion (3).

In particular, we present metrics that are based on the minimal number of inputs that must be provided to obtain a counterexample starting from states that can be reached by a given set of *logs*. We also show that, given any weight function, we may boost the performance of existing learning algorithms by introducing an extra component, which we call the *Comparator*. Preliminary experiments show that use of the Comparator yields a significant reduction of the number of inputs required to learn a correct model for the SUL, compared to a current state-of-the-art algorithm. Existing learning algorithms do not ensure that the quality of subsequent hypotheses increases. In fact, we may have $d(H', S) < d(H, S)$, even when hypothesis H is a refinement of hypothesis H'. Generalising a result of Chapter 3, we show that the quality of hypotheses never decreases when using the Comparator.

Related work

Our research is most closely related to the work presented in Chapter 3, in which we study a simple distance metric known from concurrency theory [14], in the setting of active learning. This metric is based on the minimal number of inputs required to obtain a counterexample: the longer this counterexample is, the closer a hypothesis is to the target model. Our work generalises the results of Chapter 3 to a much larger class of metrics, including log-based metrics that more accurately capture intuitive notions of quality.

Thollard et al. study the idea of bounding the distance between between the behaviour of a system and a learned model in a *passive* setting where the learning algorithm does not have the ability to interact with the system [134]. They present an algorithm for learning a probabilistic automaton from logs that trades off minimal distance (from the logs) and minimal size (of the automaton). The distance is based on the *Kullback-Leibner divergence*, which is a measure of how the learned probability distribution diverges from an expected probability distribution given by the logs [86]. Since Kullback-Leibner divergence is assymetric, it is not a distance metric and can therefore not be used in our framework.

The area of *software metrics* [129] aims to measure alternative implementations against different criteria. While software metrics mostly measure the quality of the software development process and static properties of code, our work is more ambitious since it considers the dynamic behaviour of software.

Henzinger [67] presents a general overview of work on behaviour-based metrics. Most research in this area thus far has been concerned with *directed metrics*, that is, metrics that are not required to be symmetrical. The idea is that for a given system X and requirement r, the distance function d describes the degree to which system X satisfies requirement r. Černỳ et al. for instance, define a metric that is applied to an implementation and a specification [35]. It is based on simulation relations between states of the two systems. If the specification simulates the implementation, then the distance is 0. However, if there is a state pair (q, q'), such that q does not simulate q', then a 'simulation failure game' is played. At a point where the specification has no transition with the same label as a transition of the implementation, the specification is allowed to choose some transition, at the cost of one penalty point. The distance between the two systems is then defined as the total number of penalty points reached when the implementation maximises, and the specification minimises the average number of penalty points. In our work we use metrics to compare hypotheses. Since we compare hypotheses in both directions, we use undirected (symmetric) metrics in our work. Thrane et al [135] study directed metrics between weighted automata [49]. In contrast, our work shows how weighted automata can be used to define undirected metrics between unweighted automata.

De Alfaro et al. study directed and undirected metrics in both a linear time and a branching time setting [8]. Most related to our work are the results on undirected linear distances. The starting point for the linear distances is the distance $\|\sigma - \rho\|_\infty$ between two traces σ and ρ, which measures the supremum of the differences in propositional valuations at corresponding positions of σ and ρ. The distance between two systems is then defined as the Hausdorff distance of their sets of traces. De Alfaro et al. provide a logical characterization of these distances in terms of a quantitative version of LTL, and present algorithms for computing distances over metric transition systems [8]. In particular, they present an $\mathcal{O}(n^4)$ algorithm for computing distances between states of a deterministic metric transition system, where n is the size of the structure.

The undirected linear distance metric of De Alfaro et al. does not meet our second criterion for distance metrics (existence of finite test suites). However, the authors extend their results to a discounted context in which distances occurring after i steps are multiplied by α^i, where α is a discount factor in $[0, 1]$. We expect that finite test suites do exist for the discounted metrics of De Alfaro et al. but it is not evident that the $\mathcal{O}(n^4)$ algorithm for computing distances generalizes to the metric setting.

There are intriguing relations between our results and the work of Brandán Briones et al [27] on a semantic framework for test coverage. In [27] also a general class of weight functions is introduced. These *weighted fault models* includes a finiteness condition in order to enable test coverage. However, weighted fault models do not induce a metric (since they may assign weight 0 to certain sequences) and hence the resulting theory is quite different.

4.2 Preliminaries

Sequences. Let I be any set. The set of finite sequences over I is denoted I^*. Concatenation of finite sequences is denoted by juxtaposition. We use ϵ to denote the empty sequence. The sequence containing a single element $e \in I$ is denoted as e. The length of a sequence $\sigma \in I^*$, i.e. the number of concatenated elements of σ, is denoted with $|\sigma|$. We write $\sigma \leq \rho$ to denote that σ is a prefix of ρ.

Mealy machines. We use Mealy machines as models for reactive systems.

Definition 4.1. *A Mealy machine is a tuple $M = (\Sigma, \Gamma, Q, q^0, \delta, \lambda)$, where:*

- Σ *is a nonempty, finite set of inputs,*
- Γ *is a nonempty, finite set of outputs,*
- Q *is a finite set of states,*
- $q^0 \in Q$ *is the initial state,*
- $\delta : Q \times \Sigma \to Q$ *is a transition function, and*
- $\lambda : Q \times \Sigma \to \Gamma$ *is a transition output function.*

Functions δ and λ are extended to $Q \times \Sigma^*$ by defining for all $q \in Q$, $i \in \Sigma$ and $\sigma \in \Sigma^*$,

$$\begin{aligned} \delta(q,\epsilon) &= q \quad , \qquad \delta(q, i\sigma) = \delta(\delta(q,i),\sigma), \\ \lambda(q,\epsilon) &= \epsilon \quad , \qquad \lambda(q, i\sigma) = \lambda(q,i)\lambda(\delta(q,i),\sigma). \end{aligned}$$

Observe that for each state and input pair exactly one transition is defined.

The semantics of a Mealy machine are defined in terms of output functions:

Definition 4.2. *An output function over Σ and Γ is a function $A : \Sigma^* \to \Gamma^*$ that maps each sequence of inputs to a corresponding sequence of outputs such that, for all $\sigma, \rho \in \Sigma^*$, $|\sigma| = |A(\sigma)|$, and $\sigma \leq \rho \;\Rightarrow\; A(\sigma) \leq A(\rho)$.*

Definition 4.3. *The semantics of a Mealy machine M are defined by the output function A_M given by $A_M(\sigma) = \lambda(q^0, \sigma)$, for all $\sigma \in \Sigma^*$.*

Let $M_1 = (\Sigma, \Gamma, Q_1, q_1^0, \delta_1, \lambda_1)$ and $M_2 = (\Sigma, \Gamma, Q_2, q_2^0, \delta_2, \lambda_2)$ be two Mealy machines that share common sets of input and output symbols. We say M_1 and M_2 are *equivalent*, denoted $M_1 \approx M_2$, iff $A_{M_1} = A_{M_2}$. An input sequence $\sigma \in \Sigma^*$ *distinguishes* states $q_1 \in Q_1$ and $q_2 \in Q_2$ iff $\lambda_1(q_1, \sigma) \neq \lambda_2(q_2, \sigma)$. Similarly, σ distinguishes M_1 and M_2 iff $A_{M_1}(\sigma) \neq A_{M_2}(\sigma)$.

4.3 Weight Functions and Metrics

The metrics that we consider in this chapter are parametrized by weight functions. Intuitively, a weight function specifies the importance of input sequences: the more weight an input sequence has, the more important it is that the output it generates is correct.

Definition 4.4. *A weight function for a nonempty, finite set of inputs Σ is a function $w : \Sigma^* \to \mathbb{R}^{>0}$ such that, for all $t > 0$, $\{\sigma \in \Sigma^* \mid w(\sigma) > t\}$ is finite.*

The finiteness condition in the above definition asserts that, even though the domain Σ^* is infinite, a weight function may only assign a value larger than t to a finite number of sequences, for any $t > 0$. Therefore, weight functions must involve some form of *discounting* by which long input sequences get smaller weights. This idea is based on the intuition that “a potential bug in the far-away future is less troubling than a potential bug today” [9].

Example 4.1. *Let us define a weight function w by $w(\sigma) = 2^{-|\sigma|}$, for each $\sigma \in \Sigma^*$. Let $t \in \mathbb{R}^{>0}$. In order to see that w is a weight function, observe that*

$$w(\sigma) > t \;\Leftrightarrow\; 2^{-|\sigma|} > t \;\Leftrightarrow\; |\sigma| < -\log_2 t.$$

Since Σ is finite, this implies that the set $\{\sigma \in \Sigma^* \mid w(\sigma) > t\}$ is finite, as required.

Below we define how a weight function induces a distance metric on output functions. Intuitively, the most important input sequence two output functions disagree on, i.e. the sequence with maximal weight, determines the distance.

Definition 4.5. *Let A, B be output functions over Σ and Γ, and let w be a weight function over Σ. Then the distance metric $d(A, B)$ induced by w is defined as:*

$$d(A, B) \quad = \quad \max\{w(\sigma) \mid \sigma \in \Sigma^* \wedge A(\sigma) \neq B(\sigma)\},$$

with the convention that $\max \emptyset = 0$. Sequence $\sigma \in \Sigma^*$ is a w-maximal distinguishing sequence for A and B if $A(\sigma) \neq B(\sigma)$ and $w(\sigma) = d(A, B)$.

Note that $d(A, B)$ is well-defined: the set $\{w(\sigma) \mid \sigma \in \Sigma^* \wedge A(\sigma) \neq B(\sigma)\}$ is either empty or contains, by the finiteness restriction for weight functions, a maximal element. Observe that for all output functions A and B with $A \neq B$ there exists a w-maximal distinguishing sequence.

Theorem 4.1. *Let Σ and Γ be nonempty, finite sets of inputs and outputs, and let $\mathcal{A}$ be the set of all output functions over Σ and Γ. Then the function d of Definition 4.5 is an ultrametric in the space $\mathcal{A}$ since, for any $A, B, C \in \mathcal{A}$,*

1. *$d(A, B) = 0 \Leftrightarrow A = B$ (identity of indiscernibles)*
2. *$d(A, B) = d(B, A)$ (symmetry)*
3. *$d(A, B) \leq \max(d(A, C), d(C, B))$ (strong triangle inequality)*

Proof.

1. If $A = B$ then $d(A, B) = 0$ by definition of d and the convention $\max \emptyset = 0$. We prove the converse implication by contraposition. Assume $A \neq B$. Then there exists a $\sigma \in \Sigma^*$ such that $A(\sigma) \neq B(\sigma)$. Since, by definition of w, $w(\sigma) > 0$ it follows, by definition of d, that $d(A, B) \neq 0$.

2. Follows directly from the symmetry in the definition of d.

3. We consider four cases:

   (a) If $d(A, B) = 0$ then $d(A, B) \leq \max(d(A, C), d(C, B))$ holds trivially.

   (b) If $d(A, C) = 0$ then $A = C$ by identity of indiscernibles and $d(A, B) \leq \max(d(A, C), d(C, B))$ holds trivially.

   (c) If $d(C, B) = 0$ then $C = B$ by identity of indiscernibles and $d(A, B) \leq \max(d(A, C), d(C, B))$ holds trivially.

   (d) Assume $d(A, B) \neq 0$, $d(A, C) \neq 0$ and $d(C, B) \neq 0$. Let σ_1 be a w-maximal distinguishing sequence for A and B, let σ_2 be a w-maximal distinguishing sequence for A and C, and let σ_3 be a w-maximal distinguishing sequence for C and B. Let $t_1 = w(\sigma_1)$, $t_2 = w(\sigma_2)$, and $t_3 = w(\sigma_3)$. We prove $t_1 \leq \max(t_2, t_3)$ by contradiction. Suppose $t_1 > \max(t_2, t_3)$. By definition of d, we know that for all σ with $w(\sigma) > t_2$, $A(\sigma) = C(\sigma)$. Similarly, we know that for all σ with $w(\sigma) > t_3$, $C(\sigma) = B(\sigma)$. Thus, for all σ with $w(\sigma) \geq t_1$, $A(\sigma) = C(\sigma) = B(\sigma)$. This contradicts the fact that $w(\sigma_1) = t_1$ and $A(\sigma_1) \neq B(\sigma_1)$.

□

For any weight function w, we lift the induced distance metric from output functions to Mealy machines by defining, for Mealy machines M and M', $d(M, M') = d(A_M, A_{M'})$. Observe that $d(M, M') = 0$ iff $M \approx M'$. Also, for each $\varepsilon > 0$, the set $\{\sigma \in \Sigma^* \mid w(\sigma) \geq \varepsilon\}$ is finite. Thus there exists a finite test suite that we may apply to either find a counterexample that proves $M \not\approx M'$ or to establish that $d(M, M') < \varepsilon$.

4.4 Log-Based Metrics

A *log* $\tau \in \Sigma^*$ is an input sequence that has been observed during execution of the SUL. We assume a finite set $L \subset \Sigma^*$ of logs that have been collected from the SUL. For technical reasons, we require that ϵ is included in L.

Let S be an unknown model of an SUL, and let H be a learned hypothesis for S. Since S and H are Mealy Machines, we may associate to each log $\tau \in L$ unique states $q \in Q_S$ and $q' \in Q_H$, that are reached by taking the transitions for the input symbols of τ, starting from q_S^0 and q_H^0 respectively. In this case, we say that τ *visits the state pair* (q, q'). Next, we can search for a sequence ρ that distinguishes q and q'. Now, $\tau\rho$ distinguishes q_S^0 and q_H^0, and hence $S \not\approx H$. We may define the distance of S and H in terms of the minimal number of inputs required to distinguish any pair of states (q, q') that is visited by some log $\tau \in L$.

We will now formalize the above intuition by defining a weight function and a distance metric. For this we need an auxiliary definition that describes how to decompose any trace σ into a maximal prefix that is contained in L, and a subsequent suffix.

Definition 4.6. *Let $\sigma \in \Sigma^*$ be an input sequence. An L-decomposition of σ is a pair (τ, ρ) such that $\tau \in L$ and $\tau\rho = \sigma$. We say that (τ, ρ) is a maximal L-decomposition if $|\tau|$ is maximal, i.e. for all L-decompositions (τ', ρ') of σ we have $|\tau'| \leq |\tau|$.*

Observe that, since $\epsilon \in L$, each sequence σ has a unique maximal L-decomposition (τ, ρ). We can now define the weight function w_L as a variant of the weight function of Example 4.1 in which the weight is not determined by the length of σ but rather by the length of the suffix ρ of the maximal L-decomposition.

Definition 4.7. *Let A be an output function over Σ and Γ, and let $\sigma \in \Sigma^*$. Then the weight function w_L is defined as $w_L(\sigma) = 2^{-|\rho|}$, where (τ, ρ) is the maximal L-decomposition of σ. We write d_L for the distance metric induced by w_L.*

In order to see that w_L is indeed a proper weight function in the sense of Definition 4.4, fix a $t > 0$ and derive:

$$\begin{aligned}
\{\sigma \in \Sigma^* \mid w_L(\sigma) > t\} &= \\
\{\sigma \in \Sigma^* \mid \exists \tau, \rho : 2^{-|\rho|} > t \wedge (\tau, \rho) \text{ is a maximal } L\text{-decomposition of } \sigma\} &\subseteq \\
\{\tau\rho \in \Sigma^* \mid 2^{-|\rho|} > t \wedge \tau \in L\} &= \\
\{\tau\rho \in \Sigma^* \mid \ |\rho| < -\log_2 t \wedge \tau \in L\} &
\end{aligned}$$

Since both Σ and L are finite the last set is finite, and therefore the first set is finite as well.

Observe that the metric d_L coincides with the metric presented in Chapter 3 if we take as set L of logs the singleton set $\{\epsilon\}$, as we then only take into account w-maximal distinguishing sequences starting in the initial state.

Example 4.2. *Let us illustrate our log-based metric with a simple coffee machine. The machine is always used as follows. First, a coffee pod is placed, then the machine is provided with water, then the button is pressed to obtain coffee, and finally the machine is cleaned. The logs for the coffee machine consist of the sequence* pod water button clean *and all of its proper prefixes (i.e.,* pod water button*,* pod water*,* pod*, and the empty sequence* ϵ*).*

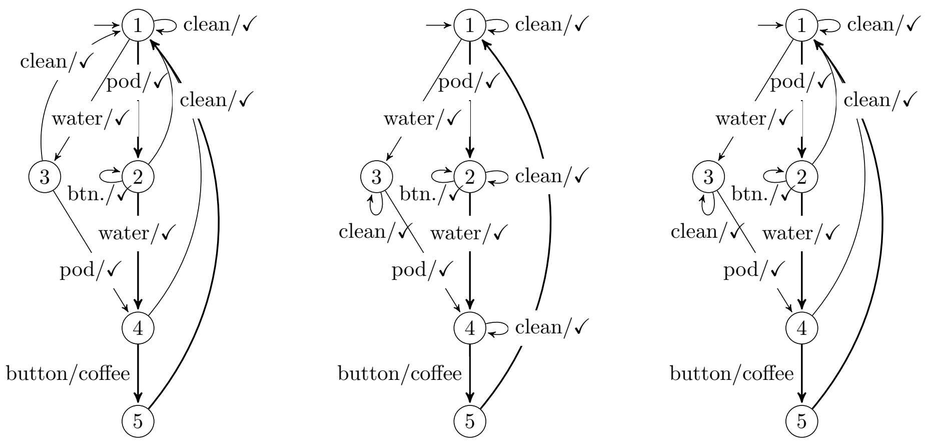
Figure 4.1 presents three models for the coffee machine. The model shown in Figure 4.1a is the correct model S, and the models shown in Figure 4.1b and 4.1c, respectively, are hypotheses H' and H for S. Observe that both hypotheses produce correct output for all logs, but that they nevertheless have some incorrect transitions. In H', the clean *transitions are incorrect in states 2, 3 and 4. In H only the erroneous* clean *transition in state 3 remains.*

Let us compute the distances of H' and H to S. A w_L-maximal distinguishing sequence to discover inequivalence of H' and S is pod water clean button*.*

At the end of this sequence, H' outputs coffee*, while S remains quiescent, i.e., output* $\checkmark$*. Despite that the sequence is of length four, it only takes two inputs to discover the error starting from a state that can be reached via a log, since state 4 is reached by* pod water*. Therefore, the distance between H' and S according to our metric is* 2^{-2}*.*

A w_L-maximal distinguishing sequence to discover the remaining error in H is water clean pod button*: H outputs* coffee *at the end of this sequence, where it should remain quiescent. Since the prefix* water *has never been observed in logs, it takes four inputs to discover this error starting from a state that has been visited by a log: state 1 is known because it is reached by the empty sequence ϵ. As a result, the distance between H and S according to our metric is* 2^{-4}*.*

Observe that these distances capture the subtle improvement in H compared to H' (as $2^{-4} < 2^{-2}$), despite that four inputs are required in both hypotheses to discover an error. Both H' and H are wrong, but the problem



(a) S (b) H' (c) H

Figure 4.1: Models of a coffee machine. The machine has one button (abbreviated as btn. in state 2), can be provided with a pod and water, and can be cleaned. It can produce coffee, or remain quiescent (✓) after an input. The logged trace is displayed with bold arrows. Some insignificant self-loops are not displayed.

with H' is more serious, as the error is visible after two transitions starting from a state that is reached during normal use of the system, instead of four transitions in H. In the metric of Chapter 3, both hypotheses would be considered equally distant to S for this reason.

Algorithm. Van den Bos [26] presents an algorithm for finding w_L-maximal distinguishing sequences for two given models. As we will see, such an algorithm is extremely useful as a component in model learning. The input of the algorithm of Van den Bos [26] consists of two Mealy machines H and H' that agree on all inputs from L, that is, $A_H(\sigma) = A_{H'}(\sigma)$, for all $\sigma \in L$. (This can be realized, for instance, by first checking for each hypothesis model whether it is consistent with all the logs in L.) The key idea of the algorithm is that minimal length distinguishing sequences (for pairs of states) are gathered by constructing a partition of indistinguishable states. By processing the partition, a distinguishing sequence of minimal length is found for each pair of states in $Q_H \times Q_{H'}$, or it is established that the states are equivalent. After that, a w_L-maximal distinguishing sequence can be found by picking a minimal length distinguishing sequence that is visited by some log in L. Intuitively, the time complexity of the search for these sequences can be deferred from the fact that a table has to be filled for all state pairs. Indeed, it follows that the algorithm is quadratic, i.e. of $\mathcal{O}(pn^2)$, where n is the sum of the number of states of H and H', and p is the number of inputs. In Chapter 2 it is shown that minimal length distinguishing sequences for all pairs of states can even be found in $\mathcal{O}(pn \log n)$.

Weighted automata. There are many possible variations of our log-based metrics. We may for instance consider variations in which the weight of a log is partially determined by its frequency. We may also assign a higher weight to logs in which certain "important" inputs occur. All such variations can be easily defined using the concept of a *weighted automaton* [49], i.e., an automaton in which states and transition carry a certain weight. Below we define a slightly restricted type of weighted automaton, called *weighted Mealy machine*, which only assigns weights to transitions.

Definition 4.8. *A weighted Mealy machine is a tuple $M = (\Sigma, \Gamma, Q, q^0, \delta, \lambda, c)$, where $(\Sigma, \Gamma, Q, q^0, \delta, \lambda)$ is a Mealy machine and $c : Q \times \Sigma \to \mathbb{R}^{>0}$ is a cost function. Cost function c is extended to $Q \times \Sigma^*$ by defining, for all $q \in Q$, $i \in \Sigma$ and $\sigma \in \Sigma^*$, $c(q, \epsilon) = 1$ and $c(q, i\sigma) = c(q, i) \cdot c(\delta(q, i), \sigma)$. The cost function $c_M : \Sigma^* \to \mathbb{R}^{>0}$ induced by M is defined as $c_M(\sigma) = c(q^0, \sigma)$.*

A cost function c_M is not always a weight function in the sense of Definition 4.4, since it may assign an unbounded weight to infinitely many sequences. However, if the weight of any *cycle* in M is less than 2 then c_M is a weight function.

Definition 4.9. *Let $M = (\Sigma, \Gamma, Q, q^0, \delta, \lambda, c)$ be a weighted Mealy machine. A path of M is an alternating sequence $\pi = q_0 i_0 q_1 \cdots q_{n-1} i_{n-1} q_n$ of states in Q and inputs in Σ, beginning and ending with a state, such that, for all $0 \leq j < n$, $\delta(q_j, i_j) = q_{j+1}$. Path π is a cycle if $q_0 = q_n$ and $n > 0$. The weight of path π is defined as the product of the weights of the contained transitions: $\prod_{j=0}^{n-1} c(q_j, i_j)$.*

Theorem 4.2. *Let M be a weighted Mealy machine, then c_M is a weight function iff all cycles have weight (strictly) less than 1.*

Let L be a prefix closed set of logs (i.e. all prefixes of a log in L are also included in L). Then the weight function w_L of Definition 4.7 can alternatively be defined as the weight function c_M induced by a weighted automaton M with states taken from $L \cup \{\perp\}$, that is, the set of logs extended with an extra sink state $\perp$, initial state ϵ, and transition function δ and cost function c defined as:

$$\delta(q,i) = \begin{cases} qi & \text{if } q \in L \wedge qi \in L \\ \perp & \text{otherwise} \end{cases} \qquad c(q,i) = \begin{cases} 1 & \text{if } q \in L \wedge qi \in L \\ \frac{1}{2} & \text{otherwise} \end{cases}$$

Note that, by Theorem 4.2, c_M is indeed a weight function.

4.5 An Adapted Learning Algorithm

In this section, we will explain how weight functions and their induced metrics can be used to improve model learning.

Active learning is a learning framework in which a Learner can ask questions (*queries*) to a Teacher, as visualized in Figure 4.2a. We assume that the Teacher is capable of answering queries correctly according to the Minimally Adequate Teacher (MAT) model of Angluin [11]. The Teacher knows a Mealy machine S which is unknown to the Learner. Initially, the Learner only knows the input and output symbols of S. The task of the Learner is to learn S by asking two types of queries:

- With a *membership query* (MQ), the Learner asks what the response is to an input sequence $\sigma \in \Sigma^*$. The Teacher answers with the output sequence $A_S(\sigma)$.
- With an *equivalence query* (EQ), the Learner asks whether a hypothesized Mealy machine H is correct, that is, whether $H \approx S$. The Teacher answers *yes* if this is the case. Otherwise it answers *no* and supplies a *counterexample*, which is a sequence $\sigma \in \Sigma^*$ that produces a different output sequence for both Mealy machines, that is, $A_H(\sigma) \neq A_S(\sigma)$.

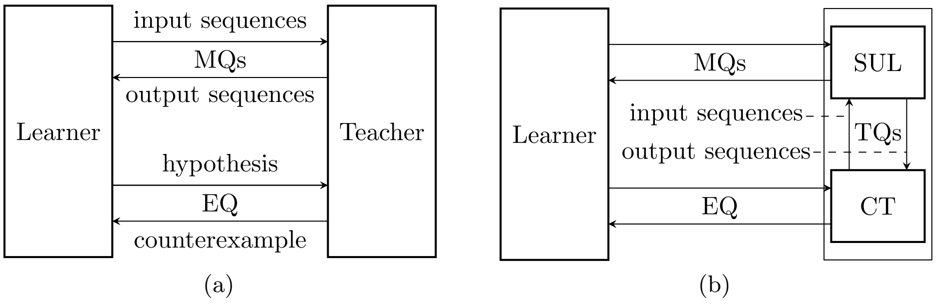


Figure 4.2: Active learning framework (a) and implementation for black-box learning (b).

Starting from Angluin's seminal L^* algorithm [11], many algorithms have been proposed for learning a Mealy machine H that is equivalent to S via a finite number of queries. We refer to [82] for an excellent recent overview. In applications in which one wants to learn a model of a black-box reactive system, the Teacher typically consists of a System Under Learning (SUL) that answers the membership queries, and a conformance testing (CT) tool [93] that approximates the equivalence queries using a set of *test queries* (TQs). A test query consists of asking the SUL what the response is to an input sequence $\sigma \in \Sigma^*$, similar to a membership query. A schematic overview of such an implementation of active learning is shown in Figure 4.2b.

We will now explain how weight functions and the metrics they induce can be used to enhance active learning. Our idea is to place a new "Comparator" component in between the Learner and the Teacher, as displayed in Figure 4.3. The Comparator ensures that the distance of subsequent hypotheses to the target model S never increases. Moreover, the Comparator may replace an equivalence query by a single membership query. This speeds up the learning process, since a Teacher typically answers an equivalence query by running a large number of test queries generated by a conformance testing algorithm. In the printer controller case study of [127], for instance, on average more than 270.000 test queries were used to implement a single equivalence query.

Assume we have a weight function w and an oracle which, for given

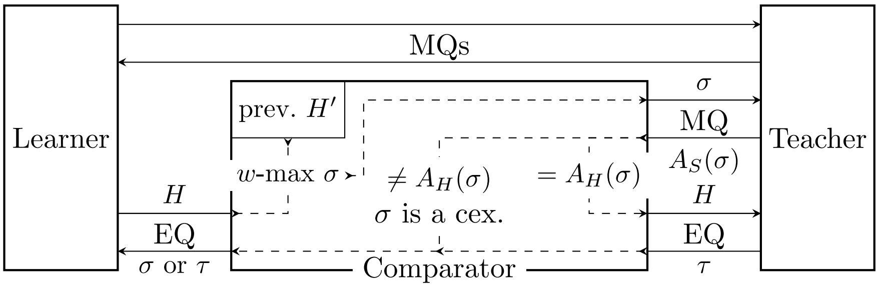


Figure 4.3: Active learning framework with the Comparator in the middle.

models H and H' with $H \not\approx H'$, produces a w-maximal distinguishing sequence, that is, a sequence $\sigma \in \Sigma^*$ with $d(H, H') = w(\sigma)$ and $A_H(\sigma) \neq A_{H'}(\sigma)$. The behavior of the Comparator component can now be described as follows:

– The first equivalence query from the Learner is forwarded to the Teacher, and the resulting reply from the Teacher is forwarded again to the Learner.

– The Comparator always remembers the last hypothesis that it has forwarded to the Teacher.

– Upon receiving any subsequent equivalence query from the Learner for the current hypothesis H, the Comparator computes a w-maximal distinguishing sequence σ for H and the previous hypothesis H', as described in the previous section. The Comparator poses a membership query σ to the Teacher and awaits the reply $A_S(\sigma)$. Depending on the reply, two things may happen:

  1. $A_S(\sigma) \neq A_H(\sigma)$. The Comparator has found a counterexamples for hypothesis H, and returns *no* together with σ to the Learner in response to the equivalence query.
  2. $A_S(\sigma) = A_H(\sigma)$. The Comparator forwards the equivalence query to the Teacher, waits for the reply, and forwards this to the Learner.

From the perspective of the Learner, the combination of a Comparator and a Teacher behave like a regular Teacher, since all membership and equivalence queries are answered appropriately and correctly. Hence the Learner will succeed to learn a correct hypothesis H after posing a finite number of queries.

Conversely, from the perspective of the Teacher, the Comparator and the Learner together behave just like a regular Learner that poses membership and equivalence queries. A key property of the Comparator/Learner combination, however, is that the quality of hypotheses never decreases. We claim that, whenever the Comparator first poses an equivalence query for H' and then for H, we always have $d(S, H) \leq d(S, H')$. In order to see why this is true, observe that when the Comparator poses the equivalence query for H it has found a w-maximal distinguishing sequence σ for H and H'. Therefore we know that $A_H(\sigma) \neq A_{H'}(\sigma)$ and

$$w(\sigma) \quad = \quad d(H', H) \tag{4.1}$$

Through a membership query σ the Comparator has also discovered that $A_S(\sigma) = A_H(\sigma)$. This implies $A_S(\sigma) \neq A_{H'}(\sigma)$ and thus

$$w(\sigma) \quad \leq \quad d(S, H') \tag{4.2}$$

Now we infer

$$\begin{array}{rcl} d(S,H) & \leq & \text{(by the strong triangle inequality, Theorem 4.1)} \\ \max(d(S,H'), d(H',H)) & = & \text{(by equation (4.1))} \\ \max(d(S,H'), w(\sigma)) & = & \text{(by inequality (4.2))} \\ d(S,H'). & & \end{array}$$

Hence, the distance between subsequent hypotheses and S never increases.

4.6 Case Studies

In this section, we present two case studies in which we measure the effect of a Comparator for the log-based metrics from Section 4.4. In the first case study, we learn a model for the Engine Status Manager (ESM), a piece of industrial software that controls the transition from one status to another in Océ printers. In the second case study we learn a model for the Windows 8 TCP server.

Engine Status Manager. In [126], a first attempt was made to learn a model for the ESM using the algorithm of Rivest & Schapire [119] as implemented in LearnLib [117]. A manually constructed reference model was used to determine the success of the learning algorithm. The author did not succeed in learning the complete model, as it took the Teacher too long to find a counterexample at some point. A second attempt was made in [127]. In this work, an adaptation of a finite state machine testing algorithm by Lee and Yannakakis [93] was used by the Teacher to find counterexamples. The authors succeeded in learning a complete, correct model with 3410 states for the ESM through a sequence of more than 100 hypotheses. Particularly because of the large number of hypotheses, this case study appeared to be a suitable case to test the impact of a Comparator.

Using the same setup for the Learner and the Teacher as in [127], we have conducted twenty independent runs for each of the following three experiments.

(a) The classical setting without a Comparator.

(b) A setting with a Comparator and the trivial log set $L = \{\epsilon\}$. This setup resembles the algorithm presented in Chapter 3.

(c) A setting with a Comparator, using the aforementioned algorithm for finding w-maximal distinguishing sequences on a nontrivial set of logs.

No real logs were available to us for setup (c), because no appropriate logging method was in place to obtain real user logs from, and setting up such logging would be tedious. Instead, we developed a method to generate logs that resemble real logs. In [126], a couple of 'paths', directly inferred from the ESM, are given. Such a path is a sequence of subsets of the input alphabet. An input sequence for the ESM can be obtained by concatenating inputs from the subsequent subsets of the path. More specifically, the algorithm for doing this keeps track of the sequence σ it has constructed, the current state q, the set of already visited states V, and the index k of the current subset of the path. Initially, $\sigma = \epsilon$, $q = q^0$, $V = \{q^0\}$, and $k = 0$. The algorithm extends σ with an input i from subset k if $\delta(q, i) \notin V$. In that case, q and V are updated accordingly. Else, we search for an input in subset $k + 1$. Only sequences with their last input in the last subset of

Table 4.1: Number of inputs used to learn a model for the ESM ($n = 20$).

setup	mean	std. dev.	median	min	max
(a)	416 519 487	219 015 166	404 307 465	209 781 273	686 385 316
(b)	371 248 375	57 005 155	377 724 597	290 072 340	545 535 231
(c)	308 928 853	50 719 369	295 863 646	243 197 179	430 523 416

Table 4.2: Number of inputs used to learn a model for a TCP server ($n = 500$).

setup	mean	std. dev.	median	min	max
(a)	263 463	254 353	206 750	35 694	2 076 538
(b)	262 948	291 222	205 487	40 927	2 380 343
(c)	259 409	241 255	210 545	41 471	2 168 348

a path are included in the logs. In total, we have generated 9800 logs for each run.

Experimental results are shown in Table 4.6. On average over 20 runs, setup (c) (with Comparator) requires 25.8% fewer inputs than setup (a) (no Comparator), and 26.8% fewer inputs than setup (b) (the algorithm of Chapter 3) to learn a correct model for the ESM. A non-parametrical, distribution independent statistical test was used to determine that this result is significant ($p < 0.05$, $z = -4.15$).

TCP. In [54], active learning was used to obtain a model for the Windows 8 TCP server. Using the aforementioned Learner and Teacher algorithms, the authors succeeded in learning a model of 38 states through a series of 13 hypotheses. We have conducted 500 runs for each of the experimental setups described above, using the model of [54] as an SUL. Experimental results are shown in Table 4.6. Unfortunately, we found no significant reduction in inputs when using the Comparator. We conjecture that this is due to the inherent simplicity of the model.

4.7 Conclusion

We have presented a general class of distance metrics on Mealy machines that may be used to formalize intuitive notions of quality. Preliminary experiments show that our metrics can be used to obtain a significant reduction of the number of inputs required to learn large black-box models. For smaller models, no reduction was found. Therefore, we conjecture that the utility of our metrics increases as models become more complex.

In future work, we plan to perform more experiments to verify these results. In addition, we wish to do experiments with real logs, instead of generated ones. Another topic for future research is to develop efficient algorithms for computing w-maximal distinguishing sequences for the weight functions induced by weighted Mealy machines. Bounding the distance between a hypothesis and the unknown target model during learning remains a challenging problem. Our experiments have produced discouraging results in this sense, since the quality of a hypothesis is hard to predict because of the high variance for different experimental runs.

Chapter 5

Model Learning as a Satisfiability Modulo Theories Problem

Rick Smetsers, Paul Fiterău-Broştean and Frits Vaandrager

Abstract

We explore an approach to model learning that is based on using *satisfiability modulo theories* (SMT) solvers. To that end, we explain how DFAs, Mealy machines and register automata, and observations of their behavior can be encoded as logic formulas. An SMT solver is then tasked with finding an assignment for such a formula, from which we can extract an automaton of minimal size. We provide an implementation of this approach which we use to conduct experiments on a series of benchmarks. These experiments address both the scalability of the approach and its performance relative to existing active learning tools.

5.1 Introduction

We are interested in algorithms that construct black-box state diagram models of software and hardware systems by observing their behavior and performing experiments. Developing such algorithms is a fundamental research problem that has been widely studied. Roughly speaking, two approaches have been pursued in the literature: *passive learning* techniques, where models are constructed from (sets of) runs of the system, and *active learning* techniques, that accomplish their task by actively doing experiments on the system.

Gold [63] showed that the passive learning problem of finding a minimal DFA that is compatible with a finite set of positive and negative examples, is NP-hard. In spite of these hardness results, many DFA identification algorithms have been developed over time, see [71] for an overview. Some of the most successful approaches translate the DFA identification problem to well-known computationally hard problems, such as SAT [69], vertex coloring [57], or SMT [108], and then use existing solvers for those problems.

Angluin [11] presented an efficient algorithm for active learning a regular language L, which assumes a *minimally adequate teacher* (MAT) that answers two types of queries about L. With a *membership query*, the algorithm asks whether or not a given word w is in L, and with an *equivalence query* it asks whether or not the language L_H of an hypothesized DFA H is equal to L. If L_H and L are different, a word in the symmetric difference of the two languages is returned. Angluin's algorithm has been successfully adapted for learning models of real-world software and hardware systems [114, 117, 140], as shown in Figure 5.1. A membership query

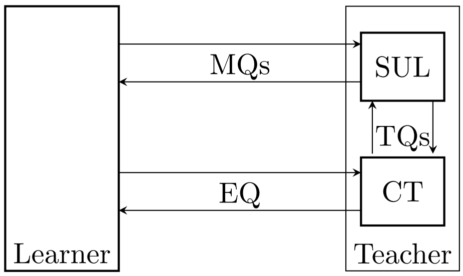


Figure 5.1: Model learning within the MAT framework.

(MQ) is implemented by bringing the *system under learning* (SUL) in its initial state and the observing the outputs generated in response to a given input sequence, and an equivalence query (EQ) is approximated using a *conformance testing tool* (CT) [93] via a finite number of *test queries* (TQ). If these test queries do not reveal a difference in the behavior of an hypothesis H and the SUL, then we assume the hypothesis model is correct.

Walkinshaw et al. [144] observed that from each passive learning algorithm one can trivially construct an active learning algorithm that only poses equivalence queries. Starting from the empty set of examples, the passive algorithm constructs a first hypothesis H_1 that is forwarded to the conformance tester. The first counterexample w_1 of the conformance tester is then used to construct a second hypothesis H_2. Next counterexamples w_1 and w_2 are used to construct hypothesis H_3, and so on, until no more counterexamples are found.

In this chapter, we compare the performance of existing active learning algorithms with passive learning algorithms that are 'activated' via the trick of Walkinshaw et al. [144]. At first, this may sound like a crazy thing to do: why would one compare an efficient active learning algorithm, polynomial in the size of the unknown state machine, with an algorithm that makes a possibly superpolynomial number of calls [12] to a solver for an NP-hard problem? The main reason is that in practical applications i/o interactions often take a significant amount of time. In [123], for instance, a case study of an interventional X-ray system is described in which a single i/o interaction may take several seconds. Therefore, the main bottleneck in these applications is the total number of membership and test queries, rather than the time required to decide which queries to perform. Also, in practical applications the state machines are often small, with at most a few dozen states (see for instance [7, 2, 123]). Therefore, even though passive learning algorithms do not scale well, there is hope that they can still handle these applications. Active learning algorithms rely on asking a large number of membership queries to construct hypotheses. Passive learning algorithms pose no membership queries, but instead need a larger number of equivalence queries, which are then approximated using test queries. A priori, it is not clear which approach performs best in terms of the total number of membership and test queries needed to learn a model.

Our experiments compare the original L* [11] and the state-of-the-art

TTT [83] active learning algorithm with an SMT-based passive learning algorithm on a number of practical benchmarks. We encode the question whether there exists a state machine with n states that is consistent with a set of observations into a logic formula, and then use the Z3 SMT solver [105] to decide whether this formula is satisfiable. By iteratively incrementing the number of states we can find a minimal state machine consistent with the observations. As equivalence oracle we use a state-of-the-art conformance testing algorithm based on adaptive distinguishing sequences [92, 128]. In line with our expectations, the passive learning approach is competitive with the active learning algorithms in terms of the number of membership and test queries needed for learning.

An advantage of SMT encodings, when compared for instance with encodings based on SAT or vertex coloring, is the expressivity of the underlying logic. In recent years, much progress has been made in extending active learning algorithms to richer classes of models, such as register automata [76, 3, 34] in which data may be tested and stored in registers. We show that the problem of finding a register automaton that is consistent with a set of observations can be expressed as an SMT problem, and compare the performance of the resulting learning algorithm with that of Tomte [3], a tool for active learning of register automata, on some simple benchmarks. New algorithms for active learning of FSMs, Mealy machines and various types of register automata are often extremely complex, and building tools implementations often takes years [3, 34, 83]. Adapting these tools to slighly different scenarios is typically a nightmare. One such scenario is when the system is missing *reset* functionality. This renders most active learning tools impractical, as these rely on the ability to reset the system. Developing SMT-based learning algorithms for register automata in settings with and without resets only took us a few weeks. This shows that the SMT-approach can be quite effective as a means for prototyping learning algorithms in various settings.

The rest of this chapter is structured as follows. Section 5.2 describes how one can encode the problem of learning a minimal consistent automaton in SMT. The scalability and effectiveness of our approach, and its applicability in practice are assessed in Section 5.3. Conclusions are presented in Section 5.4.

5.2 Model Learning as an SMT Problem

This section describes how to express this problem in a logic formula. *If and only if* there exists an assignment to the variables of this formula that makes it true, then exists an automaton A with at most n states that is consistent with S. We use an SMT solver to find such an assignment. If the SMT solver concludes that the formula is satisfiable, then its solution provides us with A.

We distinguish three types of *constraints*:

- *axioms* must be satisfied for A to behave as intended by its definition.
- *observation constraints* must be satisfied for A to be consistent with S.
- *size constraints* must be satisfied for A to have n states or less.

Hence, the problem can be solved by iteratively incrementing n until the encoding of the axioms, observation constraints and size constraints is satisfiable.

In the following subsections, we present encodings for deterministic finite automata (Section 5.2.1 and Section 5.2.2), Moore and Mealy machines (Section 5.2.3), register automata (Section 5.2.4), and input-output register automata (Section 5.2.5).

5.2.1 An Encoding for Deterministic Finite Automata

A *deterministic finite automaton* (DFA) accepts and rejects *strings*, which are sequences of labels. We define a DFA as follows:

Definition 5.1. *A DFA is a tuple (L, Q, q_0, δ, F), where*

- *L is a finite set of* labels*,*
- *Q is a finite set of* states*,*
- *$q_0 \in Q$ is the* initial state*,*
- *$\delta : Q \times L \to Q$ is a* transition function *for states and labels,*
- *$F \subseteq Q$ is a set of* accepting states*.*

Let x be a string. We use x_i to denote ith label of x. We use $x_{[i,j]}$ to denote the substring of x starting at position i and ending at position j (inclusive), i.e. $x = x_{[1,|x|]}$.

A DFA A accepts a string if its computation ends in an accepting state. This can be formalized as follows. Let $x \in L^*$ be a string, then A accepts x if a sequence of states $q'_0 \dots q'_{|x|}$ exists such that

1. $q'_0 = q_0$,
2. $q'_i = \delta(q'_{i-1}, x_i)$ for $1 \leq i \leq |x|$, and
3. $q'_{|x|} \in F$.

Let S_+ be a set of strings that should be accepted, and let S_- be a disjoint set of strings that should be rejected. Let S be the set that contains all of these strings, along with their labels, i.e. $S = \{(x, \texttt{true}) : x \in S_+\} \cup \{(x, \texttt{false}) : x \in S_-\}$. A DFA is *consistent* with S if it accepts all strings in S_+, and rejects all strings in S_-.

This leads us to a natural encoding for finding a consistent DFA in satisfiability modulo the theories of inequality and uninterpreted functions. We encode a DFA as follows:

– Q is a finite subset of the (non-negative) natural numbers $\mathbb{N}$,

– $q_0 = 0$,

– The set of accepting states F is encoded as a function $\lambda : Q \to \mathbb{B}$, such that $q \in F \iff \lambda(q) = \texttt{true}$.

The following size constraint ensures that A has at most n states:

$$\forall q \in \{0, \dots, n-1\} \quad \forall l \in L \quad \bigvee_{q'=0}^{n-1} \delta(q, l) = q' \tag{5.1}$$

Remark 5.1. *If the solver supports linear equalities, then the constraint in Equation 5.1 can be encoded more compactly as:*

$$\forall q \in \{0, \dots, n-1\} \quad \forall l \in L \quad \delta(q, l) < n \tag{5.2}$$

If we assume without loss of generality that the initial state is 0, then we can add the following constraints for the strings in S_+:

$$\forall x \in S_+ \quad \lambda(\ \delta(\dots\delta(\delta(0, x_1), x_2), \dots x_{|x|})\ ) = \texttt{true} \tag{5.3}$$

Similarly, we can add the following constraints for the strings in S_-:

$$\forall x \in S_- \quad \lambda(\ \delta(\dots\delta(\delta(0, x_1), x_2), \dots x_{|x|})\ ) = \texttt{false} \tag{5.4}$$

5.2.2 A Better Encoding for Deterministic Finite Automata

The nesting in the set of constraints given by Equation 5.3 and Equation 5.4 might lead to many redundant constraints for the theory solver, because a transition might be encoded in multiple ways. One solution to this is to define the constraints implied by strings in a non-nested way. Similarly to Heule and Verwer [69], and Bruynooghe et. al. [28], we use an *observation tree* (OT) for this. This can be considered a partial, tree-shaped automaton that is *exactly consistent* with S, i.e. it accepts only the set S_+ and rejects only the set S_-. We define an OT for a set of labeled strings in Definition 5.2.

Definition 5.2. *An OT for a set of strings $S = \{S_+, S_-\}$ is a tuple (L, Q, λ), where*

- *L is a set of labels,*
- *$Q = \{x \in L^* : x$ is a prefix of a string in $S_+ \cup S_-\}$,*
- *$\lambda : S_+ \cup S_- \to \mathbb{B}$ is a output function for the strings, with $x \in S_+ \iff \lambda(x) = \texttt{true}$.*

Now, let us explain how one can construct a set of constraints for finding a DFA $A = (L, Q^A, q_0, \delta^A, F)$ that is consistent with an OT $T = (L, Q^T, \lambda^T)$ for a given set $S = \{S_+, S_-\}$. Let us (again) consider the set of states Q^A as a set of non-negative integers with $q_0 = 0$, and let us encode the set of accepting states F as a function $\lambda : Q \to \mathbb{B}$, such that $q \in F \iff \lambda(q) = \texttt{true}$. Recall that a DFA is consistent if and only if it accepts all strings in S_+ and rejects all strings in S_-, i.e. for each x in S $\lambda^A(\delta^A(q_0, x)) = \lambda^T(x)$ (we slightly abuse notation here by extending $\delta^A : Q \times L^* \to Q$ to strings). Such a DFA has at most as many states as the

OT (but typically significantly less). Therefore, there must exist a surjective (i.e. many-to-one) function from the strings of the OT to states of the DFA:

$$map : Q^T \to Q^A \tag{5.5}$$

Our goal is to find a set of constraints for *map* that make sure that our target DFA A is consistent. For this we define the following observation constraints:

$$map(\epsilon) = q_0 \tag{5.6}$$
$$\forall xl \in Q^T : x \in L^*, l \in L \quad \delta^A(map(x), l) = map(xl) \tag{5.7}$$
$$\forall x \in S_+ \cup S_- \quad \lambda^A(map(x)) = \lambda^T(x) \tag{5.8}$$

Equation 5.6 maps the empty string to the initial state of A. Equation 5.7 encodes the observed prefixes as transitions of A while Equation 5.8 encodes the observed outputs, with λ^A encoding F.

To meet the minimality requirement, we are interested in finding the 'smallest' *map* function; i.e. there should be no function with a smaller image that satisfies these constraints. For this purpose we can re-use one of the size constraints presented earlier (Equations 5.3 or 5.2).

5.2.3 A Modification for Moore and Mealy Machines

An advantage of the encoding presented in Section 5.2.2 (as opposed to the one presented in Section 5.2.1) is that it can easily be modified to learn *transducers*. Transducers are automata that generate output strings. As such, they can be used to model input-output behaviour of software.

A *Moore machine* is a transducer that generates an output label initially and each time it (re-) enters a state. We define a Moore machine in Definition 5.3.

Definition 5.3. *A Moore machine is a tuple* $(I, O, Q, q_0, \delta, \lambda)$*, where*

- I *is a finite set of* input labels*,*
- O *is a finite set of* output labels*,*
- Q*,* q_0 *and* δ *are a set of states, the initial state, and a transition function respectively, and*

– *$\lambda : Q \to O$ is a output function that maps states to output labels.*

A set of observations S for a Moore machine consists of *traces*, which are pairs (x^I, x^O) where $x^I \in I^*$ is an *input string* and $x^O \in O^*$ is an *output string* with $|x^O| = |x^I| + 1$. A Moore machine is consistent with a set S if for each $(x^I, x^O) \in S$ it generates x^O when provided with x^I.

A *Mealy machine* is a transducer that generates an output label each time it makes a transition. We define a Mealy machine in Definition 5.4.

Definition 5.4. *A Mealy machine is a tuple $(I, O, Q, q_0, \delta, \lambda)$, where*

– *I, O, Q, q_0 and δ are the same as for a Moore machine (Definition 5.3), and*

– *$\lambda : Q \times I \to O$ is a output function that maps transitions to output labels.*

A set of observations S for a Mealy machine consists of traces (x^I, x^O) where $x^I \in I^*$ is an input string and $x^O \in O^*$ is an output string with $|x^O| = |x^I|$. Similarly to a Moore machine, a Mealy machine is consistent with a set S if for each $(x^I, x^O) \in S$ it generates x^O when provided with x^I.

It has been shown that Moore and Mealy machines are equi-expressive if we neglect the initial output label generated by a Moore machine (see e.g. [73]). Therefore, we can define an OT for a set S of traces for a Moore or Mealy machine $A = (I, O, Q^A, q_0, \delta^A, \lambda^A)$ in a similar way. We choose to define such an *input-output observation tree* (IOOT) as follows.

Definition 5.5. *An IOOT for a set of traces S is a tuple (I, O, Q, λ), where*

– *I and O are sets of input labels and output labels respecitvely,*

– *$Q = \{x \in I^* : x$ is a prefix of an input string of a trace in $S\}$,*

– *$\lambda : Q \to O$ is a output function with $\lambda(x^I_{[0,i]}) = x^O_i$ for all $(x^I, x^O) \in S$ and $1 \leq i \leq |x^I|$.*

Observe that λ is defined for all states. Also, observe that there is no need for λ to be a transition output function for Mealy machines, because there is only one string that ends in each state of an IOOT.

Let $T = (I, O, Q^T, \lambda^T)$ be an IOOT for a set of traces S, then we can determine if there is a Moore or Mealy machine A with at most n states that is consistent with S by using the set of constraints and axioms from Section 5.2.2, if we replace Equation 5.8 with Equation 5.9 (Moore machines) or Equation 5.10 (Mealy machines).

$$\forall x \in Q^T \quad \lambda^A(map(x)) = \lambda^T(x) \tag{5.9}$$

$$\forall xl \in Q^T : x \in I^*,\ l \in I \quad \lambda^A(map(x), l) = \lambda^T(xl) \tag{5.10}$$

5.2.4 An Encoding for Register Automata

DFAs and Mealy machines typically do not scale well if the domain of inputs, or the domain of data parameters for inputs, is large. The reason for this is that the semantics of the data parameters are modeled implicitly using states and transitions; inputs with different parameters are simply regarded as different inputs. A better solution is to use a richer formalism that can model them more efficiently and exploit the resulting symmetries in the state space.

A *register automaton* (RA) is such a formalism. An RA can be seen as an automaton that is extended with a set of *registers* that can store data parameters. The values in these registers can then be used to express conditions over the transitions of the automaton, or *guards*. If the guard is satisfied the transition is fired, possibly storing the provided data parameter (this is called an *assignment*) and bringing the automaton from the current *location* to the next. As such, an RA can be used to accept or reject sequences of label-value pairs. In contrast to automata without memory, the “states” in a register automaton are called locations because the *state* of the automaton also comprises the values of the registers. Therefore, an exponential number of possible states can be modeled using a small number of locations and registers.

The RAs that we define here have the following restrictions:

right invariance Transitions do not imply (in) equality of distinct registers.

no shifts Values are never moved from one register to another.

unique values Registers always store unique values.

The first two restrictions are inherent to the definition used, the third is necessary to avoid the non-determinism caused by two used registers holding the same value. While these restrictions may cause a blow-up in the number of states required to be consistent with a set of action strings [32], it has been shown that they do not affect expressivity [4, Theorem 1], i.e. for any register automaton that does not have these restrictions, there exists an equivalent register automaton in the class that we are concerned with. For a formal treatment of these restrictions and their implications, we refer to [3] and [32].

We define an RA as follows.

Definition 5.6. *An RA is a tuple $(L, R, Q, q_0, \delta, \lambda, \tau, \pi)$, where*

- *L, Q, q_0 and λ are a set of labels, a set of locations, the start location, and a location output function respectively,*
- *R is a finite set of* registers*,*
- *$\delta : Q \times L \times (R \cup \{r_\perp\}) \to Q$ is a* register transition function*,*
- *$\tau : Q \times R \to \mathbb{B}$ is a* register use predicate*, and*
- *$\pi : Q \times L \to (R \cup \{r_\perp\})$ is a* register update function*.*

We call a label-value pair an *action* and denote it $l(v)$ for input label l and parameter v. We assume without loss of generality that parameter values are integers ($\mathbb{Z}$). A sequence of actions is called an *action string*, and is denoted by σ. A set of observations S for an RA consists of action strings that should be accepted S_+, and a set of action strings that should be rejected S_-. An RA is consistent with $S = \{S_+, S_-\}$ if it accepts all action strings in S_+, and rejects all action strings in S_-.

Formally, an RA can be considered as a DFA (Definition 5.1) enriched with a finite set of registers R and two additional functions. The first function, τ, specifies which registers are in use in a location. In a location q there can be two types of transitions for a label l and parameter value v:

- If the value v is equal to some used register r, then the transition $\delta(q, l, r)$ is taken.

– Else (if the value v is different to all used registers), the *fresh* transition $\delta(q, l, r_\perp)$ is taken.

The second function, π, specifies if and where to store a value v when this fresh transition ($\delta(q, l, r_\perp)$) is taken:

– If $\pi(q, l) = r_\perp$ then the value v on transition $\delta(q, l, r_\perp)$ is not stored.

– Else (if $\pi(q, l) = r$ for some register $r \in R$), the value v on transition $\delta(q, l, r_\perp)$ is stored in register r.

Let us describe the axioms that we need for the RA to behave as intended. First, we require that no registers are used in the initial location:

$$\forall r \in R \quad \tau(q_0, r) = \texttt{false} \tag{5.11}$$

Second, if a register is used after a transition, it means that it was used before, or it was updated:

$$\begin{aligned}&\forall q \in Q \quad \forall l \in L \quad \forall r \in R \quad \forall r' \in (R \cup \{r_\perp\})\\ &\tau(\delta(q, l, r'), r) = \texttt{true} \implies \big(\, \tau(q, r) = \texttt{true} \lor \big(r' = r_\perp \land \pi(q, l) = r\big)\,\big)\end{aligned} \tag{5.12}$$

Third, if a register is updated, then it is used afterwards:

$$\forall q \in Q \quad \forall l \in L \quad \forall r \in R \quad \pi(q, l) = r \implies \tau(\delta(q, l, r_\perp), r) = \texttt{true} \tag{5.13}$$

Our goal is to learn an RA that is consistent with a set of action strings $S = \{S_+, S_-\}$. For this, we need to define a function that keeps track of the valuation of registers during runs over these action strings. Let $A = (L, R^A, Q^A, q_0, \delta^A, \lambda^A, \tau^A, \pi^A)$ be an RA, and let $T = (L \times \mathbb{Z}, Q^T, \lambda^T)$ be an OT for S. In addition to the *map* function (Equation 5.5), we define a *valuation function val* that maps a state of T and a register of A to the value that it contains:

$$\mathit{val}\colon Q^T \times R^A \to \mathbb{Z} \tag{5.14}$$

Before we construct constraints for the action strings, we *determinize* them by making them *neat* [5, Definition 7]. An action string is *neat* if each parameter value is either equal to a previous value, or equal to the largest preceding value plus one. Let a be a parameterized input, and let a(3)a(1)a(3)a(45) be an action string, then a(0)a(1)a(0)a(2) is its corresponding neat action string, for example. Aarts et al. show that in order to learn the behavior of a register automaton it suffices to study its neat action strings, since any other action string can be obtained from a neat one via a zero respecting automorphism [5, Section 5].

Constructing constraints for an RA is a bit more involving than for the formalisms that we have discussed so far. First, we map empty string to the initial location of A (Equation 5.6). Second, we assert that a register is updated if its valuation changes, and that it is not updated if it keeps its value:

$$\forall \sigma l(v) \in Q^T \quad \forall r \in R^A \qquad \mathit{val}(\sigma l(v), r) \neq \mathit{val}(\sigma, r) \implies \pi^A(\mathit{map}(\sigma), l) = r \tag{5.15}$$

$$\forall \sigma l(v) \in Q^T \quad \forall r \in R^A \qquad \mathit{val}(\sigma l(v), r) = \mathit{val}(\sigma, r) \implies \pi^A(\mathit{map}(\sigma), l) \neq r \tag{5.16}$$

Additionally, we assert the inverse (i.e. that a register's valuation changes if and only if it is updated):

$$\forall \sigma l(v) \in Q^T \quad \forall r \in R^A \qquad \mathit{val}(\sigma l(v), r) = \begin{cases} v & \text{if } \delta^A(\mathit{map}(\sigma), l, r_\perp) = \mathit{map}(\sigma l(v)) \\ & \quad \wedge\ \pi(\mathit{map}(\sigma), l) = r \\ \mathit{val}(\sigma, r) & \text{otherwise} \end{cases} \tag{5.17}$$

Third, we encode the observed transitions:

$$\forall \sigma l(v) \in Q^T \qquad \mathit{map}(\sigma l(v)) = \begin{cases} \delta^A(\mathit{map}(\sigma), l, r) & \text{if } \exists! r \in R : \tau^A(\mathit{map}(\sigma), r) = \mathtt{true} \\ & \quad \wedge\ \mathit{val}(\sigma, r) = v \\ \delta^A(\mathit{map}(\sigma), l, r_\perp) & \text{otherwise} \end{cases} \tag{5.18}$$

Finally, we encode the observed outputs. This can be done in the same way as for DFAs (see Equation 5.8).

The task for the SMT solver is to find a solution that is consistent with these constraints. Obviously, we are interested in an RA with the minimal number of locations and registers. The number of locations can be limited in the same way as states were limited for DFAs (see Equation 5.1 or Equation 5.2). The number of registers is defined by the variables r that we quantify over in the presented equations. Therefore, they can be limited as such. In our case, the number of registers is never higher than the number of locations (because we can only update a single register from each location). Hence, the learning problem can be solved iteratively incrementing the number of locations n, and for each n incrementing the number of registers from 1 to n, until a satisfiable encoding is found.

5.2.5 An Extension for IO Register Automata

An *input-output register automaton* (IORA) is a register automaton transducer that generates an output action (i.e. label and value) after each input action. As in the RA-case, we restrict both input and output labels to a single parameter. Input and output values may update registers. Input values may be tested for (dis-)equality with values in registers. Output values can be equal to the stored values, or may be fresh. As such, an input-output register automaton can be used for modeling software that produces parameterized outputs.

For a formal description of IORAs we refer to [5]. We define an IORA in Definition 5.7. Again, in the interest of our encoding, our definition is very different from that in [5]. Despite this, the semantics are similar.

Definition 5.7. *An IORA is a tuple* $(I, O, R, Q, q_0, \delta, \lambda, \tau, \pi, \omega)$*, where*

- I *and* O *are finite, disjoint sets of input and output labels,*
- R*,* Q*,* q_0*,* τ *and* π *are the same as for an RA (Definition 5.6),*
- $\delta : (Q \cup \{q_\perp\}) \times (I \cup O) \times (R \cup \{r_\perp\}) \rightarrow (Q \cup \{q_\perp\})$ *is a* register transition function with a sink location*,*
- $\lambda : (Q \cup \{q_\perp\}) \rightarrow \mathbb{B}$ *is a* location output function with a sink location*, and*

– $\omega : Q \to \mathbb{B}$ *is a* location type function *that returns* `true` *if a location is an* input location*, and* `false` *if it is an* output location*.*

A set of observations S for an IORA consists of *action traces*, which are pairs (σ^I, σ^O) where $\sigma^I \in (I \times \mathbb{Z})^*$ is an *input action string*, and $\sigma^O \in (O \times \mathbb{Z})^*$ is an *output action string* with $|\sigma^I| = |\sigma^O|$. An IORA is consistent with a set S if for each pair $(\sigma^I, \sigma^O) \in S$ it generates σ^O when provided with σ^I.

Despite that semantically an IORA is a transducer, we define it as an RA (Definition 5.6) which distinguishes between input and output labels, and which defines an additional function ω for the location type. From an *input location* transitions are allowed only for input actions. After an input action the IORA reaches an *output location*, in which a *single* transition is allowed. This transition determines the output action generated in response, as well as the input location the IORA will transition to. Transitions that are not allowed lead to a designated *sink location*, which is denoted $q_\perp$.

Using this definition allows us to incorporate the axioms defined for our RA encoding (Equations 5.11–5.13) also in our IORA encoding. To these, we add the following axioms for an IORA to behave as intended.

First, observe that we do not use λ as an output function for an IORA. Instead, we use it to denote which locations are allowed. Hence, we require that the sink location $q_\perp$ is the only rejecting location:

$$\forall q \in (Q \cup \{q_\perp\}) \quad \lambda(q) = \begin{cases} \texttt{false} & \text{if } q = q_\perp \\ \texttt{true} & \text{otherwise} \end{cases} \tag{5.19}$$

Second, we require that transitions do not lead to the sink location:

$$\forall q \in Q \quad \forall o \in O \quad \forall r \in (R \cup \{r_\perp\}) \quad \omega(q) = \texttt{true} \implies \delta(q, o, r) = q_\perp \tag{5.20}$$

$$\forall q \in Q \quad \forall i \in I \quad \forall r \in (R \cup \{r_\perp\}) \quad \omega(q) = \texttt{false} \implies \delta(q, i, r) = q_\perp \tag{5.21}$$

$$\forall l \in I \cup O \quad \forall r \in (R \cup \{r_\perp\}) \quad \delta(q_\perp, l, r) = q_\perp \tag{5.22}$$

Finally, we require that input locations are *input enabled* (Equation 5.23), and that there is there is only one transition possible in an output location (Equation 5.24):

$$\forall q \in Q \quad \forall i \in I \quad \forall r \in (R \cup \{r_\perp\}) \quad \omega(q) = \texttt{true} \implies \delta(q, i, r) \neq q_\perp \tag{5.23}$$

$$\forall q \in Q \quad \exists! o \in O \quad \exists! r \in (R \cup \{r_\perp\}) \quad \omega(q) = \texttt{false} \implies \delta(q, o, r) \neq q_\perp \tag{5.24}$$

Our goal is to learn an IORA $A = (I, O, R^A, Q^A, q_0, \delta^A, \lambda^A, \tau^A, \pi^A, \omega^A)$ that is consistent with a set of action traces S. Because of the nature of our encoding, we consider each action trace $\sigma = (\sigma^I, \sigma^O)$ in S as an interleaving of the input action string σ^I and the output action string σ^O, i.e. $\sigma = \sigma^I_1 \sigma^O_1 \ldots \sigma^I_{|\sigma^I|} \sigma^O_{|\sigma^I|}$. Let $T = ((I \cup O) \times \mathbb{Z}, Q^T, \lambda^T)$ be an OT for such strings.

The constraints for an IORA can now be constructed in the same way as for an RA (Equation 5.6 and Equation 5.15–5.18). Observe that we do not use λ to encode the observed outputs (this is already done by encoding the transitions of the OT). Instead, λ is used to denote which locations are allowed. All the locations in Q are allowed (because we have observed them) and $q_\perp$ is the only location that is not allowed ($\lambda(q_\perp) = \texttt{false}$ by Equation 5.19). As such, we add the following constraint:

$$\forall \sigma \in Q^T \quad \mathit{map}(\sigma) \neq q_\perp \tag{5.25}$$

We can now determine if there is an IORA with at most n locations and m registers in the same way as for RAs, i.e. by iteratively incrementing the number of locations n, and for each n incrementing the number of registers from 1 to n, until a satisfiable encoding is found.

5.3 Implementation and Evaluation

We implemented our encodings using Z3Py, the Python front-end of Z3 [105][1]. Our tool can generate an automaton model from a given set of observations (passive learning), or a reference to the system and a tester implementation (active learning), also when this system cannot be reset. We have also implemented a tester for the classes of automata supported. The tester generates test queries (or *tests*) each test consisting of an access sequence to an arbitrary state in the current hypothesis, and a sequence generated by a random walk from that state. In experiments, we configure the tester to build shorter tests. Longer tests worsen the scalability of our tool, and are unneeded for learning small models. All experimental results shown

[1]See `https://gitlab.science.ru.nl/rick/z3gi/tree/lata`

were obtained using our most efficient encodings, namely, those involving an OT and not relying on linear equalities (all other encodings performed considerably worse in terms of scalability).Validity of the learned models was ensured by running a large number of tests on the last hypothesis and checking the number of states. We conducted a series of experiments to assess the scalability and effectiveness of our approach.

Our first experiment assesses the scalability of our encodings by adapting the scalable *Login*, *FIFO set* and *Stack* benchmarks of [3] to DFAs, Mealy machines, RAs and IORAs.

The systems benchmarked are IORA by nature, and are parameterizable by their size (the maximum number of registered users or the size of the collection). The systems only generate OK and NOK as output, with no parameters. This facilitates the generation of RA/Mealy/DFA representations by applying an adapter over the system, exposing an interface corresponding to the respective formalism. The RA adapter, for example, accepts a sequence of inputs only if all outputs generated by the system are OK, otherwise it rejects the sequence. Our adaptation made the Mealy and DFA versions of FIFO set and Stack systems equivalent, hence we only consider FIFO sets for these formalisms. The Login systems are simplified by removing the password parameter (so `login`, `register` and `logout` are done solely by supplying a user id), as our implementation does not yet support actions with multiple parameters.

To generate tests, we used the testing algorithm described earlier. The maximum length of the random sequence is $3 + size$, where $size$ is the number of users or elements in the system. The *solver timeout* – the amount of time the solver was provided to compute a solution or indicate its absence – was set to 10 seconds for the DFA and Mealy systems, and to 10 minutes for the RA and IORA systems whose constraints could take considerably longer to process.

We initially terminated learning runs whenever the SMT solver failed to return a result within this time bound (or the SMT solver *timed out*). We then realized that even in cases where the SMT solver timed out, it might still find a solution in a subsequent iteration (for a greater n). This solution might not be minimal, but it was nevertheless consistent with past observations. We thus allowed each learning run to iterate until an upper bound was reached.

Table 5.1: Scalable experiments. The model name encodes the formalism, type and size.

Model	succ	states	regs	tests		inputs		time(sec)	
		loc		avg	std	avg	std	avg	std
DFA_FIFOSet(1)	5	3.0		17.0	8.28	53.0	36.35	0.44	0.07
DFA_FIFOSet(2)	5	4.0		19.0	10.33	80.0	45.58	0.68	0.11
DFA_FIFOSet(3)	5	5.0		28.0	12.71	141.0	69.99	1.83	0.46
DFA_FIFOSet(4)	5	6.0		48.0	41.12	294.0	293.18	3.44	0.42
DFA_FIFOSet(5)	5	7.0		108.0	19.62	788.0	161.23	7.24	1.54
DFA_FIFOSet(6)	5	8.0		125.0	42.06	953.0	361.76	22.34	9.03
DFA_FIFOSet(7)	5	9.0		136.0	34.52	1126.0	344.18	28.55	12.65
DFA_FIFOSet(8)	5	10.0		228.0	81.11	2156.0	832.02	76.28	42.49
DFA_FIFOSet(9)	2	11.5		413.5	161.93	4194.0	2104.35	199.99	26.1
DFA_Login(1)	5	4.0		100.0	30.8	432.0	140.46	3.06	0.52
DFA_Login(2)	5	7.0		167.0	95.4	932.0	618.15	14.94	2.54
DFA_Login(3)	5	11.0		446.0	100.18	3092.0	781.73	131.84	39.26
Mealy_FIFOSet(1)	5	2.0		4.0	1.1	14.0	5.12	0.13	0.0
Mealy_FIFOSet(2)	5	3.0		9.0	2.51	39.0	12.54	0.49	0.09
Mealy_FIFOSet(3)	5	4.0		16.0	5.32	90.0	30.96	0.71	0.07
Mealy_FIFOSet(4)	5	5.0		14.0	8.64	108.0	62.35	1.44	0.64
Mealy_FIFOSet(5)	5	6.0		24.0	11.03	166.0	86.08	1.96	0.27
Mealy_FIFOSet(6)	5	7.0		36.0	14.85	307.0	151.59	3.81	0.8
Mealy_FIFOSet(7)	5	8.0		41.0	14.38	373.0	138.2	9.7	2.57
Mealy_FIFOSet(8)	5	9.0		90.0	26.53	928.0	310.48	20.87	2.73

Mealy_FIFOSet(9)	5	10.0		131.0	22.68	1547.0	296.28	34.64	5.67
Mealy_FIFOSet(10)	5	11.0		162.0	66.45	1948.0	787.28	60.08	12.93
Mealy_FIFOSet(11)	5	12.0		280.0	110.65	3694.0	1722.57	79.75	23.69
Mealy_FIFOSet(12)	4	15.5		370.0	200.12	5021.0	3312.85	227.15	290.99
Mealy_FIFOSet(13)	2	14.5		526.5	318.91	8021.5	5608.06	190.36	51.96
Mealy_Login(1)	5	3.0		12.0	5.45	52.0	21.43	0.78	0.07
Mealy_Login(2)	5	6.0		44.0	12.15	264.0	83.27	6.37	1.09
Mealy_Login(3)	5	10.0		104.0	10.03	726.0	69.93	52.4	4.83
Mealy_Login(4)	1	16.0		241.0	0.0	2094.0	0.0	370.19	0.0
RA_Stack(1)	5	3.0	1	32.0	21.18	109.0	90.48	3.18	0.86
RA_Stack(2)	5	5.0	2	202.0	71.88	1018.0	394.58	124.72	53.41
RA_FIFOSet(1)	5	3.0	1	49.0	12.92	180.0	52.56	4.94	6.07
RA_FIFOSet(2)	5	6.0	2	365.0	88.41	2025.0	578.33	333.09	334.12
RA_Login(1)	5	4.0	1	306.0	163.18	1336.0	765.23	54.96	9.99
RA_Login(2)	3	8.0	2	1606.0	345.22	9579.0	2163.67	6258.11	1179.27
IORA_Stack(1)	5	7.0	1	7.0	1.58	24.0	6.63	8.77	1.92
IORA_FIFOSet(1)	5	7.0	1	8.0	3.27	31.0	9.36	6.45	0.84
IORA_Login(1)	5	9.0	1	33.0	6.65	152.0	29.89	1509.18	477.04

For each system we performed 5 learning runs and collated the resulting statistics.

The results are shown in Table 5.1. Columns describe the system, the number of successful learning runs, the number of states/locations (which may vary due to loss of minimality) and registers (where applicable), average and standard deviation for the number of tests and inputs used in learning except for validating the last hypothesis, and for the amount of time learning took. The table only includes entries for systems we could learn.

In our second experiment we used our tool to learn simulated models obtained by the learning case studies described in [7, 2, 123]. These models are Mealy machines detailing aspects of the behavior of bankcard protocols, biometric passports and power control services (PCS). For the purpose of this experiment we connected the open-source tester used in [128][2]. This produces tests similar to our own, but extended by distinguishing sequences. These tests are parameterized by both the length of the random sequence and a factor k. We set both the length and the k factor to 1. We note that our simple tester (which doesn't append distinguishing sequences) could not reliably found counterexamples for several of these models. We attribute this to the large size of the models' input alphabets. The solver timeout was set to 1 minute.

Results are shown in Table 5.2. Columns are as in the previous experiment, with an additional column used to describe the size of the input alphabet. Our approach is able to learn all models, though it takes a considerable amount of time for the larger models. There are no cases where we cannot learn the model.

Our third experiment pits our approach against LearnLib (v0.12.1) [82] and Tomte (v0.41) [3]. LearnLib is a known FSM learning framework, while Tomte is a learner for IORAs. Both LearnLib and Tomte are configured to use TTT, a state-of-the-art learning algorithm within Angluin's framework. LearnLib is additionally configured to use the original L* learning algorithm. The setups for all learners use caching to ensure that only tests uncovering new observations are included in statistics. We compare our approach to LearnLib on both the scalable and case study models, and to Tomte on the scalable models. The testers are the same as in previous experiments. Due to the high standard deviation, we ran 20

[2]See `https://gitlab.science.ru.nl/moerman/Yannakakis`

Table 5.2: Case study experiments.

Model	succ	states	alph	tests		inputs		time(sec)	
			size	avg	std	avg	std	avg	std
Biometric Passport	5	6	9	173.0	90.75	848.0	574.57	28.85	3.82
MAESTRO	5	6	14	1159.0	280.78	6193.0	1690.15	330.87	15.33
MasterCard	5	6	14	703.0	192.03	3560.0	1133.96	337.44	80.17
PIN	5	6	14	767.0	188.76	3825.0	1095.39	328.0	41.85
SecureCode	5	4	14	290.0	67.33	1340.0	318.29	82.25	29.46
VISA	5	9	14	839.0	169.53	5005.0	1161.63	1933.03	498.98
PCS_1	5	8	9	704.0	178.94	3861.0	1123.0	201.74	19.68
PCS_2	5	3	9	72.0	7.96	284.0	22.01	8.89	1.3
PCS_3	5	7	9	555.0	175.55	2973.0	1078.96	146.89	21.89
PCS_4	5	7	9	583.0	224.07	3029.0	1626.2	158.33	18.66
PCS_5	5	9	9	1158.0	163.37	6218.0	1165.34	750.83	135.16
PCS_6	5	9	9	778.0	517.2	4087.0	3204.23	735.55	75.77

experiments for each benchmark.

A comparison between the learners is drawn in Table 5.3. Our approach needs fewer tests than L*. However, it requires more inputs on several of the PCS case study models. This can be attributed to L* being able to learn these systems without processing any counterexamples. By contrast, L* severely lags behind on the scalable systems benchmarks, which require a series of counterexamples. Our approach also largely defeats TTT on these benchmarks, and even on some of the case study models. Our approach defeats Tomte on all (admittedly very basic) models. In summary, although the approach appears less effective than TTT, it is still competitive and mostly outmatches Tomte and L*.

A reason to why our approach performs worse than TTT on the case study models may have to do with how hypotheses are constructed. Hypotheses constructed by TTT are completed in terms of their output behavior by running new tests. In contrast, our approach constructs hypotheses solely on the basis of counterexamples. For states whose output behavior has not yet been covered by counterexamples, the solver just produces a *guess* which is likely wrong. This may decrease the efficacy of test algorithms which actively use outputs to compute distinguishing sequences (as does the algorithm used in the case study models).

We remark that the seemingly better results we achieved on the scalable systems may be due to their simplistic nature. Having only few inputs, these systems don't benefit as much from the smart exploratory tests a learner may execute.

Although the sample size is small, results seem to indicate that whereas FSM learners are efficient, active register automata learners are yet to reach this level of optimization. These learners often resort to expensive counterexample analysis procedures in order to simplify the counterexample, as in shortening it or isolating the relevant data relations. This simplification is needed in order to minimize the counterexample's subsequent impact on the performance of learning. By contrast, our approach does not need such procedure. One should note however, that the models our approach can learn lack succintness (they are unique valued and non-swapping). Consequently, the number of tests may be adversely affected by the number of registers of a system.

Our final experiment assesses our extension for learning systems without

Table 5.3: Comparison with other learners

Model	states	alph		SMT		TTT		L*		Tomte	
	loc	size	tests	inputs	time	tests	inputs	tests	inputs	tests	inputs
Biometric Passport	6	9	220	1057	26	220	941	333	1143		
MAESTRO	6	14	835	4375	359	860	4437	1190	4718		
MasterCard	6	14	839	4379	353	996	5260	1190	4718		
PIN	6	14	757	3945	338	911	4769	1190	4718		
SecureCode	4	14	313	1485	90	194	682	798	2758		
VISA	9	14	796	4770	2115	750	4094	2040	9015		
PCS_1	8	9	629	3530	189	417	2179	657	2682		
PCS_2	3	9	71	279	9	75	196	252	657		
PCS_3	7	9	508	2651	154	476	2472	576	2196		
PCS_4	7	9	559	3024	154	451	2297	576	2196		
PCS_5	9	9	1120	6260	778	417	1753	1308	5340		
PCS_6	9	9	1158	6442	704	457	1977	1308	5340		
Mealy_FIFOSet(2)	3	2	6	27	0	12	38	14	38		
Mealy_FIFOSet(7)	8	2	52	481	7	71	588	235	2494		
Mealy_FIFOSet(10)	11	2	179	2152	63	163	1822	486	6743		
Mealy_Login(2)	6	3	37	214	7	57	242	57	219		
Mealy_Login(3)	10	3	89	644	64	120	704	240	1720		
IORA_FIFOSet(1)	7	2	9	31	7					21	36.5
IORA_Stack(1)	7	2	8.5	33	8					19	34
IORA_Login(1)	9	3	33	152	849					157	580

Table 5.4: Learning models without resets

States	succ	inputs		time(sec)	
		avg	std	avg	std
1	5	2.0	0.0	0.03	0.0
2	5	6.0	9.55	0.13	0.07
3	5	21.0	6.4	0.43	0.11
4	5	47.0	32.9	0.88	0.21
5	5	48.0	21.48	1.89	0.6
6	5	76.0	53.18	12.95	7.7
7	2	71.5	10.61	25.65	2.59
8	1	288.0	0.0	106.35	0.0

resets using benchmarks from recent related work[115]. These benchmarks involve learning randomly generated Mealy machines of increasing size with 2 input labels and 2 output labels. These models are connected though they may not be minimal. We adapted our random walk algorithm for setting without resets, using a fixed random length of 3. The solver timeout was set to 10 seconds. Table 5.4 illustrates results. Our extension performs and scales worse than the approach in [115] and does not provide any guarantees of correctness. However, being able to learn such systems by a simple extension showcases the versatility of an SMT-based approach.

Notes on scalability Scalability is the main weakness of our approach. The IORA and RA encodings scaled up to only a maximum of size 2 for stacks and FIFO sets. By comparison, Tomte can learn FIFO sets of size 30 [4]. The Mealy machine encoding scaled up to a size of 13 for the FIFO set, besting the DFA encoding, which only managed 9. Learning can take several minutes due to the large number of times the solver has to be called. Our implementation calls the solver on every new counterexample, and there can be hundreds of counterexamples in a learning run.

We note some of the measures adopted towards improving scalability while maintaining simplicity of encodings. These measures largely pertain to how the definitions were implemented into Z3Py. We initially defined sorts for states, locations, labels, inputs and outputs using *Datatype* constructs, which provide a natural representation for expressing sets. We later found that defining a custom sort by using *DeclaredSort* was more efficient (the Mealy machine and DFA FIFO set examples could only scale to 10 and 6 states respectively when using the *Datatype* formulation). This optimization was used for the DFA and Mealy machine encodings. We also found that it was often useful to avoid universal quantifiers (and instead expand them to quantifier-free formulas), in particular when formulating constraints over nodes. Finally, the time the solver took to provide a solution increased with the size in nodes of the OT. Configuring the test algorithm to generate tests with a random sequence of size 2 meant the VISA model could not be reliably learned with a solver timeout of 1 minute.

While some measures where taken to improve scalability, we can definitely see room for improvement. In particular, the IORA encodings could be made a lot more efficient by utilizing a more succint underlying definition. The current definition requires roughly a doubling of the number of locations, as well as a function to distinguish between input and output locations. A more succint definition would use a transducer-style output function, with each transition encoding both input and output semantics. Another hindering factor is that we still use *Datatype* constructs for implementing both RA and IORA encodings.

5.4 Conclusion

We have experimented with an approach for model learning which uses SMT solvers. The approach is highly versatile, as shown in its adaptations for learning FSMs and register automata, and for learning without resets. We provide an open source tool implementing these adaptations. Experiments indicate that our approach is competitive with the state-of-the-art. While the approach does not scale well, we have shown that it can be used for learning small models in practice. In the future we wish to improve the scalability of the approach via more efficient encodings. We hope this chapter gives rise to a broader direction of future work, since the presented approach has several advantages over traditional model learning algorithms. Notably, it appears to be quite effective for rapid prototyping of learning algorithms for new formalisms and settings.

Chapter 6

Complementing Model Learning with Mutation-Based Fuzzing

Rick Smetsers, Joshua Moerman, Mark Janssen and Sicco Verwer

Abstract

An ongoing challenge for learning algorithms formulated in the MAT framework is to efficiently obtain counterexamples. In this chapter we compare and combine conformance testing and mutation-based fuzzing methods for obtaining counterexamples when learning finite state machine models for the reactive software systems of the Rigorous Exampination of Reactive Systems (RERS) challenge. We have found that for the LTL problems of the challenge the fuzzer provided an independent confirmation that the learning process had been successful, since no additional counterexamples were found. For the reachability problems of the challenge, however, the fuzzer discovered more reachable error states than the learner and tester, albeit in some cases the learner and tester found some that were not discovered by the fuzzer. This leads us to believe that these orthogonal approaches are complementary in the context of model learning.

6.1 Introduction

Software systems are becoming increasingly complex. *Model learning* is quickly becoming a popular technique for reverse engineering such systems. Instead of viewing a system via its internal structure, model learning algorithms construct a formal model from observations of a system's behaviour.

One prominent approach for model learning is described in a seminal paper by Angluin [11]. In this work, she proved that one can effectively learn a model that describes the behaviour of a system if a so-called *minimally adequate teacher* is available. This teacher is assumed to answer two types of questions about the (to the learner unknown) *target*:

- In a *membership query* (MQ) the learner asks for the system's output in response to a sequence of inputs.
- In an *equivalence query* (EQ) the learner asks if its hypothesis is equivalent to the target. If this is not the case, the teacher provides a *counterexample*, which is an input sequence that distinguishes the hypothesis and the target.

The learner iteratively asks membership queries to construct an hypothesis, and then uses a counterexample from an equivalence query to refine the hypothesis. This process iterates until the learner's hypothesis is equivalent to the target.

Peled et al. have recognized the avail of Angluin's work for learning models of real-world, reactive systems that can be modeled by a finite state machine (FSM) [114]. Membership queries, on the one hand, are implemented simply by interacting with the system. Equivalence queries, on the other hand, require a more elaborate approach, as there is no trivial way of implementing them. Therefore, an ongoing challenge, and the topic of this chapter, is to efficiently obtain counterexamples.

Several techniques for obtaining counterexamples have been proposed. The most widely studied approach for this purpose is *conformance testing* [47]. In the context of learning, the goal of conformance testing is to establish an equivalence relation between the current hypothesis and the target. This is done by posing a set of so-called *test queries* to the system. In a test query, similarly to a membership query, the learner asks for the system's response to a sequence of inputs. If the system's response is the same as the predicted response (by the hypothesis) for all test queries, then the hypothesis is assumed to be equivalent to the target. Otherwise, if there is a test for which the target and the hypothesis produce different outputs, then this input sequence can be used as a counterexample.

One of the main advantages of using conformance testing is that it can distinguish the hypothesis from all other finite state machines of size at most m, where m is a user-selected bound on the number of states. This means that if we know a bound m for the size of the system we learn, we are guaranteed to find a counterexample if there exists one. Unfortunately, conformance testing has some notable drawbacks. First, it is hard (or even impossible) in practice to determine an upper-bound on the number of states of the system's target FSM. Second, it is known that testing becomes exponentially more expensive for higher values of m [142]. Therefore, the learner might incorrectly assume that its hypothesis is correct. This motivates the search for alternative techniques for implementing equivalence queries.

The field of *mutation-based fuzzing* provides opportunities here. In essence, *fuzzers* are programs that apply a test (i.e. input sequence) to a target program, and then iteratively modify this sequence to monitor whether or not something interesting happens (e.g. crash, different output, increased code coverage . . .). Fuzzers are mostly used for security purposes, as a crash could uncover an exploitable buffer overflow, for example. *Mutation-based* fuzzers randomly replace or append some inputs to the test query.

Recently, good results have been achieved by combining mutation-based fuzzing with a genetic (evolutionary) algorithm. This requires a fitness function to evaluate the performance of newly generated test query, i.e. a measurement of 'how interesting' it is. In our case, this fitness function is based on what code is executed for a certain test query. The fittest test cases can then be used as a source for mutation-based fuzzing. Hence, tests are mutated to see if the coverage of the program is increased. Iterating this process creates an evolutionary approach which proves to be very effective for various applications [155].

RERS Challenge 2016

In this chapter we describe our experiments in which we apply the aforementioned techniques to the *Rigorous Examination of Reactive Systems*

(RERS) challenge 2016. The RERS challenge consists of two parts:

1. *problems* (i.e. reactive software) for which one has to prove or disprove certain logical properties, and
2. problems for which one has to find the reachable error states.

In our approach, we have used a state-of-the-art learning algorithm (*learner*) in combination with a conformance testing algorithm (*tester*) to learn models for the RERS 2016 problems. In addition, we have used a mutation-based fuzzing tool (*fuzzer*) to generate potentially interesting traces independently of the learner and the tester. We have used these traces as a verification for the learned models and found that

- For part (1) of the challenge the fuzzer did not find any additional counterexamples for the learner, compared to those found by the tester. Therefore the fuzzer provided an independent confirmation that the learning process had been successful.
- For part (2) of the challenge the fuzzer discovered more reachable error states than the learner and tester, albeit in some cases the learner and tester found some that were not discovered by the fuzzer.

Our experiments lead us to believe that in some applications, fuzzing is a viable technique for finding *additional* counterexamples for a learning setup. In this chapter, in addition to describing our experimental setup for RERS in detail, we therefore describe possible ways of combining learning and fuzzing.

Related work

Duchene et al. have used a combination of model learning and evolutionary mutation-based fuzzing to detect web injection vulnerabilities [51]. Their function to evaluate the fitness of a test is specific to XSS attacks, however. This makes their approach less applicable to a more general class of software systems.

Cho et al. have proposed to use *concolic testing* for implementing equivalence queries [39]. Concolic testing uses a combination of symbolic and concrete execution to build a model of the system, and then uses the model to guide further exploration of the state-space. Similarly to our fuzzing approach, the aim of concolic testing is to generate tests that maximize code coverage. The authors use a priority queue to prioritize concrete sequences that are used for symbolic execution. The sequences that visit a larger number of new basic blocks, unexplored by prior sequences, have higher priority. In contrast to our approach, however, it uses a constraint solver to evaluate the fitness of a test, and therefore requires the system to be a *white-box*. If such a white-box system is available, then concolic testing would arguably outperform fuzzing. Our fuzzing approach, however, can be applied to learn systems for which only a binary executable is available, as it can use runtime instrumentation to measure the fitness of a test. This makes our approach applicable to a broader class of software systems.

Similarly, Giannakopoulou et al. have introduced a framework that combines model learning with symbolic execution to automatically generate models for white-box components that include methods with parameters [60]. The transitions of the generated models are labeled with method names and guarded with constraints on the corresponding method parameters. The guards partition the input spaces of parameters, and enable a more precise characterization of legal orderings of method calls. The performance bottleneck faced by this approach was that the set of test sequences used to implement an equivalence query grew exponentially. Moreover, to handle methods with parameters completely, each sequence in this set was fully explored symbolically, which further increased the cost of the approach. As a consequence, the approach could only be used to learn relatively small models. To counteract this, Howar et al. presented a new framework based on that of Giannakopoulou et al., with novel algorithms for performing most of the work at a concrete level [75].

Bertolino et al. have suggested to use *monitoring* for implementing equivalence queries, i.e. continuously validating the current hypothesis against traces obtained by observing the system while it is running [19]. This approach was proposed as a focus point of the Emergent Connectors for Eternal Software Intensive Networked Systems (CONNECT) European project. Whilst such an approach of 'life-long learning' seems interesting for some applications, its utility is (to the best of our knowledge) yet to be proven in a convincing case study. A major limitation of this approach is that the quality of the learned model heavily relies on the diversity of the traces that are (passively) being observed.

6.2 Preliminaries

In this section, we describe preliminaries on finite state machines, model learning, conformance testing, and fuzzing.

6.2.1 Finite State Machines

A *finite state machine* (FSM) is a model of computation that can be used to design computer programs. At any time, a FSM is in one of its (finite number of) states, called the *current state*. Generally, the current state of a computer program is determined by the contents of the memory locations (i.e. variables) that it currently has access to, and the values of its registers, in particular the program counter. Changes in state are triggered by an event or condition, and are called *transitions*. We assume that transitions are triggered based on events, or *inputs*, that can be observed.

Formally, we define a FSM as a Mealy machine $M = (I, O, Q, q_M, \delta, \lambda)$, where I, O and Q are finite sets of *inputs*, *outputs* and *states* respectively, $q_M \in Q$ is the *start state*, $\delta : Q \times I \to Q$ is a *transition function*, and $\lambda : Q \times I \to O$ is an *output function*. The functions δ and λ are naturally extended to $\delta : Q \times I^* \to Q$ and $\lambda : Q \times I^* \to O^*$. Observe that a FSM is deterministic and input-enabled (i.e. complete) by definition.

For $q \in Q$, we use $\lfloor q \rfloor_M$ to denote a *representative* access sequence of q, i.e. $\delta(q_M, \lfloor q \rfloor_M) = q$. We extend this notation to arbitrary sequences, allowing to transform them into representative access sequences: for $x \in I^*$, we define $\lfloor x \rfloor_M = \lfloor \delta(q_M, x) \rfloor_M$.

A *discriminator* for a pair of states q, q' is an input sequence $x \in I^*$ such that $\lambda(q, x) \neq \lambda(q', x)$.

The behaviour of a FSM M is defined by a *characterization function* $A_M : I^* \to O^*$ with $A_M(x) = \lambda(q_M, x)$ for $x \in I^*$. FSMs M and M' are *equivalent* if $A_M(x) = A_{M'}(x)$ for $x \in I^*$.

6.2.2 Model Learning

The goal of so-called active model learning algorithms is to learn a FSM $H = (I, O, Q_H, q_H, \delta_H, \lambda_H)$ for a system whose behaviour can be characterized by a (unknown) FSM $M = (I, O, Q_M, q_M, \delta_M, \lambda_M)$, given the set of inputs I and access to the characterization function A_M of M.

The *TTT algorithm* is a novel model learning algorithm [83]. Its distinguishing characteristic is its redundancy-free handling of counterexamples. The TTT algorithm maintains a prefix-closed set S of access sequences to states. These states correspond to leaves of a *discrimination tree* T, in which the inner nodes are labeled with elements from a suffix-closed set of discriminators E, and its transitions are labeled with an output.

A hypothesis is constructed by *sifting* the sequences in $S \cdot I$ through the discrimination tree: Given a prefix ua, with $u \in S$ and $a \in I$, starting at the root of T, at each inner node labelled with a discriminator $v \in E$ a membership query $A_M(uav)$ is posed. Depending on the last output of this query, we move on to the respective child of the inner node. This process is repeated until a leaf is reached. The state in the label of the leaf becomes the target for transition $\delta(\delta(q_H, u), a)$.

The way that the TTT algorithm handles counterexamples is based on the observation by Rivest and Shapire that a counterexample $x \in I^*$ can be decomposed in a prefix $u \in I^*$, input $a \in I$, and suffix $v \in I^*$ such that $x = uav$ and $A_M(\lfloor u \rfloor_H av) \neq A_M(\lfloor ua \rfloor_H v)$ [120]. Such a decomposition shows that the state $q = \delta_H(\delta_H(q_H, u), a)$ is incorrect, and that this transition should instead point to a new state q' with access sequence $\lfloor u \rfloor_H a$. Therefore, this sequence is added to S. Observe that this does not affect the prefix-closedness of S. In the discrimination tree T, the leaf corresponding to q is replaced with an inner node labelled by the *temporary* discriminator v. A technique known as *discriminator finalization* is applied to construct the subtree of this newly created inner node, and obtain a minimal discriminator for q and q'. For a description of discriminator finalization, we refer to [83].

6.2.3 Conformance Testing

Conformance testing for FSMs is an efficient way of finding counterexamples. Let $H = (I, O, Q_H, q_H, \delta_H, \lambda_H)$ be a hypothesis with n states. We call a conformance testing method m-complete if it can identify the hypothesis in the set of all FSMs with at most m states. Such m-complete methods are generally polynomially in the size of the hypothesis and exponential in $m-n$, which are far more efficient than an exhaustive search. For an overview of some m-complete methods, we refer to [47]. All of these methods require the following information:

– A set of *access sequences* $S = \{\lfloor q \rfloor_H | q \in Q_H\}$, possibly extended to a *transition cover* set $S \cdot I$.

– A *traversal set* I^l that contains all input sequences of length $l = m - n + 1$, where $m = |Q_M|$ and $n = |Q_H|$.

– A means of pairwise distinguishing all states of H, such as set of *discriminators* E for all pairs of states in H.

A test suite is then constructed by combining these sets, or subsets of these sets, e.g. $S \cdot I^l \cdot E$. The difference between different testing methods is how states are distinguished (i.e. the last part).

In the so-called *partial W-method*, or *Wp-method*, [58] states are distinguished pairwise: For each state $q \in Q_H$ a set $E_q \subset E$ of discriminators is constructed, such that for each state $q' \in Q \setminus \{q\}$ there is a sequence $w \in E_q$ that distinguishes q and q', i.e. $\lambda_H(q, w) \neq \lambda_H(q', w)$. Then, each trace $uv, u \in S \cdot I, v \in I^l$ is extended with the set E_q where $q = \delta_H(q_H, uv)$.

Conformance testing is typically expensive due to the exponential size of the traversal set. Given a hypothesis H with n states and k inputs, the worst-case length of a test suite (i.e. the sum of the length of all sequences) is of order $\mathcal{O}(k^l n^3)$ (recall that $l = m - n + 1$, where m is the upper bound on the number of states of M). Moreover, it is hard to estimate an upper bound for M in practice. Often a value close to n is picked for m, in the hope that the test set contains at least some counterexamples. Then, for each next hypothesis, m is assumed to be larger than before, and eventually one hopes to obtain a correct estimate for m.

In some cases, however, the shortest counterexample for the current hypothesis is simply too long. Therefore, alternative methods for finding counterexamples might yield better results.

6.2.4 Fuzzing

A *mutation-based fuzzer* is a program that applies a set of tests (i.e. input sequences) to a target program, and then iteratively mutates these tests to monitor if ‘something interesting’ happens. This could be a crash of the target program, a change in its output, or it finds that more code is covered (via instrumentation). The *American Fuzzy Lop* (AFL) fuzzer [155] is interesting for its approach in combining mutation-based test case generation with *code coverage* monitoring.

AFL supports programs written in C, C++, or Objective C and there are variants that allow to fuzz programs written in Python, Go, Rust or OCaml. AFL works on instrumented binaries of these programs, and supports compile-time or runtime instrumentation. The tool is bundled with a modified version of gcc (afl-gcc) that can add instrumentation at compile time. The compile-time instrumentation has the best performance, but requires the source code of the target program to be available. When the source code is not available, AFL applies runtime instrumentation, which uses emulation (QEMU or Intel Pin) to achieve the result. This, however, is 2-5× slower than compile-time instrumentation [155].

From a high-level, simplified perspective, AFL works by taking a program and a queue of tests, and iteratively mutating these tests to see if the *coverage* of the program is increased; new tests that increase coverage are added to the queue. An overview of this algorithm is shown in Algorithm 8. In the next paragraphs, we will describe in more detail how coverage is measured by AFL, which mutation strategies are applied, and how execution time is minimized.

Algorithm 7: High-level overview of AFL

```
Input: a program and a queue of initial test cases
loop
    take next test case from the queue
    forall available mutation strategies do
        mutate test case
        run target program on test case
        if coverage is increased then
            add new test case to the queue
```

Measuring coverage

In order to measure coverage, AFL uses instrumentation of the control flow of the program (branches, jumps, etc.), to identify which parts of the target program are used in a given test. Using this knowledge, AFL can decide which test cases cover behaviour not previously seen in other test cases.

Internally, coverage is measured by using a so-called *trace bitmap*, which is a 64 kB array of memory shared between the fuzzer and the instrumented target. This array is updated by the following code every time an edge in the control flow is taken.

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Every location in the array is represented by a compile-time random value. When an edge in the control flow is taken, the bitmap is updated at the position of the current location and an xor of the previous location value. The intention is that every edge in the control flow is mapped to a different byte in the bitmap.

Note that because the size of the bitmap is finite and the values that represent locations in the code are random, the bitmap is probabilistic: there is a chance that collisions will occur. This is especially the case when the bitmap fills up, which can happen when fuzzing large programs with many edges in their control flow. AFL can detect and resolve this situation by applying instrumentation on fewer edges in the target or by increasing the size of the bitmap.

Mutation strategies

At the core of AFL is its ‘engine’ to generate new test cases. As mentioned earlier, AFL uses a collection of techniques to mutate existing test cases into new ones, starting with basic deterministic techniques and progressing onto more complex ones. The author of AFL has described the following strategies [153]:

- Performing sequential, ordered *bit flips* to a sequence of one, two, or four bits of the input.

– An extension of bit flips to (a sequence of one, two or four) *bytes*.

– Applying *simple arithmetic* (incrementing and decrementing) to integers in the input.

– *Overwriting* integers in the input by values from set of pre-set integers (such as -1, 1024 and MAX_INT), that are known to trigger edge conditions in many programs.

– When the deterministic strategies (above) are exhausted, randomised *stacked operations* can be applied, i.e. a sequence of single-bit flips, setting discovered byte values, addition and subtraction, inserting new random single-byte sets, deletion of blocks, duplication of blocks through overwrite or insertion, and zeroing blocks.

– The last-resort strategy involves taking two known inputs from the queue that cover different code paths and *splicing* them in a random location.

Fork server

In general, fuzzers generate a lot of tests. Therefore, many invocations of the target process are required. Instead of starting a new process for every test, AFL uses a *fork server* to speed up fuzzing. The fork server initialises the target process only once, and then forks (clones) it to create a new instance for each test case.

On modern operating systems, a process fork is done in a copy-on-write fashion, which means that any memory allocated by the process is only copied when it is modified by the new instance. This eliminates most (slow) memory operations compared to a regular process start [154], and allows for an execution of approximately 10 000 tests per second on a single core of our machine.

6.3 Experimental Setup

In this section we describe the experiments in which we apply the aforementioned techniques to the *Rigorous Examination of Reactive Systems* (RERS) challenge 2016. The RERS challenge consists of two parts:

1. A set of nine *problems* (i.e. reactive software), numbered 1 through 9, for which one has to prove or disprove a set of given *linear temproal logic* (LTL) formulae, and
2. a set of nine problems, numbered 10 through 18, for which one has to determine whether or not a set of error statements present in the source code are *reachable*, and provide a sequence of inputs such that the error statement is executed.

In our approach, we have used a state-of-the-art learner in combination with a tester to learn FSMs for the RERS 2016 problems. In addition, we have used a fuzzer to generate potentially interesting traces independently of the learner and the tester. As we have executed the learner/tester and the fuzzer indepenedently of one another, we describe their experimental setup and result in turn. The code for our experiments is available at `https://gitlab.science.ru.nl/moerman/rers-2016/`.

6.3.1 Learning and Testing with LearnLib

For our learning and testing experiments, we have used LearnLib, an open-source Java library for active model learning [102]. As a learner in LearnLib consideres its system under learning as a black-box, we have interfaced LearnLib with a compiled binary of each of the 18 problems. Below, we list and explain the choices we have made regarding our LearnLib setup.

Learning algorithm For our learning algorithm, we have chosen the TTT algoritm as implemented in LearnLib, because previous experiments have shown that it scales up to larger systems under learning; both in the amount of membership queries asked and in the amount of memory used in the process.

Testing algorithm For our testing algorithm, we have used our own implementation of the Wp method. Recall that the Wp method in principle generates a test suite whose size is polynomial in the size of the hypothesis and exponential in the upper bound of states in the system, minus the size of the hypothesis. Instead of exhausting this test suite, our implementation of the algorithm randomly samples test sequences until it finds a counterexample: First, it samples a prefix uniformly from the state-cover set of the current hypothesis. Then, it randomly generates an infix over all inputs according to a geometric distribution. Finally, we sample a suffix uniformly from the set of state-specific discriminators. By using a geometric distribution for the infix, we are not bounding the length of the test sequence. In our tool, the minimal and expected length of the infix can be set by parameters. In our experiments, its minimal length was three, and its expected length was eleven.

Counterexample handling Counterexamples were processed using the LinearForward handler in LearnLib.

Cache We have used the cache that is implemented in LearnLib to avoid sending duplicate queries to the system under learning.

The final hypothesis for each of the problems was stored as a DOT file. In order to solve the LTL formulae for part (1) of the challenge, these DOT files were translated to NuSMV. For part (2), it sufficed to `grep` the DOT files for the unique outputs that were generated in an error state.

6.3.2 Fuzzing with AFL

Independently of learning and testing the challenge's problems, we have used AFL to fuzz them. Below, we give an overview of some of the details of our experimental setup.

Instrumentation We have used the `afl-gcc` compiler that comes bundled with AFL to compile the C source code for each of the problems. This compiler instruments the control flow of the program, and implements the fork server.

Input alphabet AFL requires an input alphabet as a source for its mutation strategies. We have used the valid inputs that were defined in the source code for each problem as an input alphabet.

Error handling In order to compile the reachability problems (10 - 18) an external error handling function had to be provided. This function is called with a unique identifier whenever an error state is reached. Our implementation of the error function prints the unique error identifier, and then aborts the program. This way, each trace whose execution leads to an error state is registered by AFL as a crash. As these traces are stored in a separate results folder by AFL, we could easily separate them from traces that did not lead to an error state.

Post-processing As the input bytes that AFL considers are not limited to the valid inputs for the challenge problems, we filtered out the bytes that were not accepted.

The traces that were found by AFL were simulated on the final hypothesis of the learner to see if its output differed from that of the program binary.

6.4 Results

The results for the learning/testing setup described in Section 6.3.1 are shown in Section 6.1 and Section 6.2.

We are confident that the learned models for the LTL problems (1–9) are complete, as the last hypothesis was learnt within 1 day and no further counterexamples were found in the following week. The same holds for the first of the reachability problems (10). Beware, however, that we can never guarantee completeness with black-box techniques.

For problems 11–18 we know that we do *not* have complete models, as the learner was still finding new states every 10 minutes when the server rebooted for maintenance. The learner ran for a bit more than 7 days and saved all hypotheses. As a result of this reboot, we do not have statistics on the number of queries.

The results for the fuzzing setup described in Section 6.3.2 are shown in Table 6.3 and Table 6.4. These results should be interpreted as follows:

cycles The number of times the fuzzer went over all the interesting test traces discovered, fuzzed them, and looped back to the very beginning.

execs The total number of test traces executed.

paths The total number of test traces found that have a unique execution path.

Table 6.1: Learning and testing results for the LTL problems of RERS 2016 on an Intel(R) Xeon(R) CPU E7-4870 v2 @ 2.30GHz (server), with Oracle Java 8 JVM configured with a 40GB heap.

size	plain	arithmetic	data structures
small	**Problem 1** time: 50s states: 13 hypotheses: 6	**Problem 2** time: 1m22s states: 22 hypotheses: 10	**Problem 3** time: 7m05s states: 26 hypotheses: 13
medium	**Problem 4** time: 34m states: 157 hypotheses: 77	**Problem 5** time: 2h43m states: 121 hypotheses: 50	**Problem 6** time: 4h51m states: 238 hypotheses: 156
large	**Problem 7** time: 11h45m states: 610 hypotheses: 407	**Problem 8** time: 24h22m states: 646 hypotheses: 432	**Problem 9** time: 18h31m states: 854 hypotheses: 550

Table 6.2: Learning and testing results for the reachability problems of RERS 2016 on an Intel(R) Xeon(R) CPU E7-4870 v2 @ 2.30GHz (server), with Oracle Java 8 JVM configured with a 40GB heap.

size	plain	arithmetic	data structures
small	**Problem 10** time: 2m39s states: 59 hypotheses: 3	**Problem 11** time: 1w+ states: 22 589 hypotheses: 8 314	**Problem 12** time: 1w+ states: 12 771 hypotheses: 4 325
medium	**Problem 13** time: 1w+ states: 12 848 hypotheses: 5 564	**Problem 14** time: 1w+ states: 11 632 hypotheses: 4 513	**Problem 15** time: 1w+ states: 7 821 hypotheses: 3 792
large	**Problem 16** time: 1w+ states: 8 425 hypotheses: 3 865	**Problem 17** time: 1w+ states: 11 758 hypotheses: 5 584	**Problem 18** time: 1w+ states: 8 863 hypotheses: 4 246

Table 6.3: Fuzzing results for the LTL problems of RERS 2016 on a Intel(R) Xeon(R) CPU E7-4870 v2 @ 2.30GHz (server). The fuzzer was terminated after approximately 10 days.

size	plain	arithmetic	data structures
small	**Problem 1** cycles: 46 521 execs: 2.64×10^9 paths: 253	**Problem 2** cycles: 30 088 execs: 2.68×10^9 paths: 480	**Problem 3** cycles: 19 551 execs: 2.52×10^9 paths: 453
medium	**Problem 4** cycles: 460 execs: 8.68×10^8 paths: 3 453	**Problem 5** cycles: 4 191 execs: 2.63×10^9 paths: 1 115	**Problem 6** cycles: 109 execs: 7.32×10^8 paths: 4 494
large	**Problem 7** cycles: 35 execs: 6.86×10^8 paths: 9 556	**Problem 8** cycles: 16 execs: 6.75×10^8 paths: 10 906	**Problem 9** cycles: 71 execs: 7.63×10^8 paths: 11 305

Table 6.4: Fuzzing results for the reachability problems of RERS 2016 on a Intel(R) Xeon(R) CPU E7-4870 v2 @ 2.30GHz (server). The fuzzer was terminated after approximately 10 days.

size	plain	arithmetic	data structures
small	**Problem 10** cycles: 70 336 execs: 2.58×10^9 paths: 139	**Problem 11** cycles: 10 365 execs: 2.34×10^9 paths: 801	**Problem 12** cycles: 5 971 execs: 2.14×10^9 paths: 1 032
medium	**Problem 13** cycles: 779 execs: 2.35×10^9 paths: 4 235	**Problem 14** cycles: 621 execs: 2.02×10^9 paths: 3 838	**Problem 15** cycles: 1 040 execs: 2.77×10^9 paths: 3 685
large	**Problem 16** cycles: 50 execs: 7.22×10^8 paths: 11 908	**Problem 17** cycles: 19 execs: 4.58×10^8 paths: 10 283	**Problem 18** cycles: 21 execs: 4.58×10^8 paths: 10 237

Table 6.5: Number of error states found.

size	plain	arithmetic	data structures
small	**Problem 10**	**Problem 11**	**Problem 12**
	learner: 45 (0)	learner: 20 (0)	learner: 21 (0)
	fuzzer: 45 (0)	fuzzer: 22 (2)	fuzzer: 21 (0)
	total: 45	total: 22	total: 21
medium	**Problem 13**	**Problem 14**	**Problem 15**
	learner: 28 (0)	learner: 27 (0)	learner: 27 (0)
	fuzzer: 30 (2)	fuzzer: 30 (3)	fuzzer: 32 (5)
	total: 30	total: 30	total: 32
large	**Problem 16**	**Problem 17**	**Problem 18**
	learner: 29 (1)	learner: 27 (1)	learner: 28 (0)
	fuzzer: 31 (3)	fuzzer: 28 (2)	fuzzer: 32 (4)
	total: 32	total: 29	total: 32

For the LTL problems of the challenge, none of the test traces that have a unique execution path were counterexamples for the last hypothesis of the learner. This, in combination with the large number of cycles completed by the fuzzer, strengthens our belief that the learned models for these problems (1 - 9) are complete.

The number of reachable error states found by the learner and the fuzzer are shown in Table 6.5. The first entry in each cell is the number of unique error states that were found, and the second entry is the number of error states that were found by the given technique, but were not found by the other technique (e.g. “fuzzing: 28 (2)” means that the fuzzer has found 28 error states, and 2 of those were not found by the learner).

From these results we conclude that the fuzzer discovered more reachable error states than the learner/tester, albeit in some cases the learner/tester found some that were not discovered by the fuzzer.

6.5 Present and Future Work

The goal of our present and future research in this area is to combine model learning and mutation-based fuzzing in the following ways.

1. use fuzzing as a source of counterexamples *during* learning, and
2. use (intermediate) learning results to guide mutation-based fuzzing.

At this point in time, we have already put some significant effort into (1): Most importantly, we have implemented a new equivalence oracle, AFLEQOracle, in LearnLib, which iteratively loads a traces that AFL marks as interesting, and parses them as a test query for the learner. Unfortunately, we were unable to apply this new equivalence oracle to the RERS challenge due to time restrictions. The code for this project is available at `https://github.com/praseodym/learning-fuzzing`.

In this section we give an overview of our current effort on using mutation-based fuzzing as a source of counterexamples during learning.

An overview of the architecture for combining AFL and LearnLib is shown in Figure 6.1. To establish this, we had to tackle the following main issues:

– As AFL is provided as a standalone tool, we have created a library, libafl, that the learner can communicate with.

– As LearnLib is written in Java, and AFL (and libafl) are written in C, we needed to bridge all communication between the two. For this purpose, we have used the Java Native Interface (JNI) programming interface, which is part of the Java language. JNI allows for code running in the Java Virtual Machine (i.e. LearnLib) to interface with platform-specific native binaries or external libraries (i.e. libafl).

– We have added the possibility to embed the target program in AFL's fork server. For each membership or test query, the fork server creates a new instance of the target process. This speeds up the execution of learning, independent of the technique used to find counterexamples.

There were some other issues that we had to address:

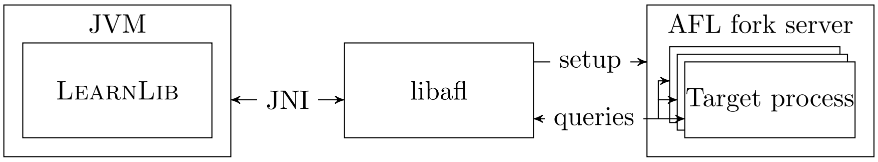


Figure 6.1: Architecture for combining LearnLib and AFL.

- AFL is designed such that it does not care about the target program's output. Instead only coverage data is used as a measure for test case relevancy. The learner, however, relies on output behaviour. Therefore, we have extended AFL to always save data from the target's stdout into a shared memory buffer (shared between libafl and the fork server process). The content of this shared memory buffer is returned to LearnLib after a successful query.
- AFL runs the target program in a non-interactive manner, i.e. it provides the program with input once and then expects it to terminate and reset state. This is in contrast to the default behaviour of LearnLib, which expects a *single-step system under learning* that repeatedly accepts an input value and returns the associated output, and has an explicit option to reset. We initially simulated this behaviour in AFL by running the target program once for each prefix of an input sequence. For the RERS challenge, however, we could run each input sequence once, as it was easy to correlate individual inputs to their corresponding outputs.

We have performed some inital experiments with the setup described above. In these experiments we compared different learning setups on their ability for finding error states in the reachability problems of the RERS 2015. For these problems, the number of reachable error states are now known.

A selection of the results is shown in Table 6.6. In addition to the number of (reachability) states learned, this table compares learning performance in terms of learning time and the number of queries needed (lower is better). In all cases, using fuzzing equivalence delivers models with more states and more reachability states found in a shorter learning time. One remark here is that the learning time we report only includes the time the learning process ran, not the time that the fuzzer ran. We ran the AFL fuzzer on

Table 6.6: Results for the RERS 2015 challenge problems on a Intel Xeon CPU E5-2430 v2 @ 2.50GHz (virtualised server), with Oracle Java 8 JVM configured with 4GB heap.

problem	method	states	errors	time	queries
1	TTT, W-method 1	25	19/29	4s	7 342
1	L*, W-method 8	25	19/29	13h	2.46×10^8
1	TTT, fuzzing	334	29/29	21s	16 731
1	**L*, fuzzing**	1 027	29/29	44m	2.86×10^6
2	TTT, W-method 1	188	15/30	1h	8.15×10^6
2	L*, W-method 3	195	15/30	17h	2.39×10^7
2	TTT, fuzzing	2 985	24/30	13m	412 340
2	**L*, fuzzing**	3 281	24/30	13h	4.21×10^7
3	L*, W-method 1	798	16/32	110h	2.42×10^9
3	TTT, fuzzing	1 054	19/32	13m	698 409
3	**L*, fuzzing**	1 094	19/32	13h	2.34×10^7
4	TTT, W-method 7	21	1/23	4h	5.17×10^7
4	**TTT, fuzzing**	7 402	21/23	16m	458 763
5	L*, W-method 1	183	15/30	13h	2.20×10^6
5	**TTT, fuzzing**	3 376	24/30	8m	416 943
6	L*, W-method 1	671	16/32	93h	8.89×10^8
6	**TTT, fuzzing**	3 909	23/32	45m	2.80×10^6

each problem for one day, and the test cases that were generated during that time were used for equivalence testing using the learning process.

6.6 Conclusion

An ongoing challenge for learning algorithms formulated in the Minimally Adequate Teacher framework is to efficiently obtain counterexamples. In this chapter we have compared and combined conformance testing and mutation-based fuzzing methods for obtaining counterexamples when learning finite state machine models for the reactive software systems of the RERS challenge. We have found that for the LTL problems of the challenge the fuzzer did not find any additional counterexamples for the learner, compared to those found by the tester. For the reachability problems of the challenge, however, the fuzzer discovered more reachable error states than the learner and tester, albeit in some cases the learner and tester found some that were not discovered by the fuzzer. This leads us to believe that in some applications, fuzzing is a viable technique for finding additional counterexamples for a learning setup.

Chapter 7

Protocol Message Format Inference and its Applications in Security

Rick Smetsers, Joeri de Ruiter, Sicco Verwer, and Erik Poll

Abstract

A promising application of model learning is in the area of *protocol inference.* Protocol inference refers to some automated form of reverse engineering the workings of a communication protocol. This can be useful for security analysis in different ways. It can be used to reverse-engineer unknown protocols, to detect security flaws in implementations of known protocols, to fingerprint implementations, or to detect anomalies in protocol usage, for example.

A prerequisite for using model learning in this area is that the protocol's *message format* (i.e. input format) is known. In this chapter we give an overview of tools and techniques for inferring the protocol message format, and their applications in security.

Chapter 7

7.1 Introduction

Protocols play a crucial role in modern-day IT systems. They are used between parties that communicate across a network (the prototypical example being TCP/IP), between different hardware components (e.g., USB), between processes on the same machine (for example OS services, such as a CUPS printer service), and between different components within one process (e.g., the protocols provided by APIs).

Protocols are of great importance for the security of these systems, as any interface at which a system can be attacked comes with an associated protocol. An attacker can try to exploit security flaws in the protocol itself or in a particular implementation of the protocol. For protocols that involve cryptography, such flaws may be of cryptographic nature, but more often than not they are more mundane implementation mistakes, such as the SSL Goto bug on Apple iOS or the Heartbleed bug in OpenSSL.

Even if different implementations of the same protocol do not contain exploitable flaws, they can (and often do) exhibit differences in behaviour. This is often the case because there is some form of freedom or ambiguity in the protocol's specification (if such a specification exists at all), or simply because the implementations contain mistakes. As a result, an implementation may have unique characteristics that provide a *fingerprint* of that implementation. Such fingerprints can be interesting for attackers, as it leaks information about the system they are targeting.

The characteristics of the usage of a protocol may also be used as a basis for *anomaly detection*. Deviations from the normal protocol usage may indicate malicious intent, and can thus be used for intrusion detection.

Attackers not only try to attack the protocols that their victims use, but they may also use their own protocols as part of their attacks. The prime example here is that controlling a botnet requires some communications protocol between the bots and the command and control centre. *Reverse engineering* such protocols, and ideally finding security flaws, may be useful to take botnets down.

This importance of protocols has motivated a need for formalisms in which different protocol implementations and specifications can be described in a similar way. One way to achieve this is by viewing a protocol implementation not via its internal structure, but through the laws which govern its behaviour: which input messages does it accept at which point, and which messages does it produce in response? This way, implementations (and specifications) that exhibit completely dissimilar compositions, for example because they are written in different programming languages, can still be characterized and analysed through the same set of rules.

There are typically two levels at which one can formally describe the behaviour, or 'language', of a protocol: the *protocol message format* and the *protocol state machine*. The former describes the structure for individual valid messages in the protocol and the latter describes the temporal control-specific behaviour and data dependencies of messages that make up a protocol session.

In this chapter, we give an overview of tools and techniques for inferring the protocol message format and their applications in security, with the aim to bridge the gap between the academic and applied world, and propagate further research in the area. It was observed by Bossert and Guilhéry that there is a huge difference between the academic and the applied world in the field of protocol inference for security applications [24]. In the applied world on the one hand, "experts can be [seen] as fighters specialized in one-lines commands [that are] able to compute any CRC and format trans-coding by heart". In the academic world on the other hand, papers emerge in different subfields of (software) engineering, that use different terminology for similar problems related to protocol inference. This has resulted in a dissonance between researchers and security experts.

Recently, two other survey papers have appeared on the subject of protocol inference [106, 52]. Both of these surveys explicitly focus on the *tools* that have been proposed. Our survey contains such an overview as well, but in addition gives a comprehensive and cohesive overview of the *techniques* that these tools implement. We believe that this approach is more useful in bridging the gap between the academic and applied world, and propagate further research in the area.

Overview Most of the work on protocol inference focuses on communications protocols (despite that the techniques can be applied elsewhere as well). Therefore, we first introduce the terminology of communications protocols in Section 7.2. Then, in Section 7.3, we give a general classification of protocol inference techniques. In Section 7.4 we describe the techniques that have been proposed for reverse engineering message formats. In Section 7.5 we give an overview of the applications of these approaches in security. Finally, in Section 7.6 we conclude our work.

7.2 Terminology

A (communications) *protocol* is a set of rules that two (or more) parties use to communicate. It defines the structure and intent of the exchange of *messages*. A message is an atomic piece of the interaction. In most practical cases, a message is defined as a sequence of bits going in the same communication direction (i.e. going from one party to the other). The structure of a message is determined by its *message format*, typically as a collection of *message fields*. A message field is a sequence of consecutive bits with some specified meaning. The *values* that a message field can take are captured by its *domain*. The domain for the sequence number field in the TCP protocol, for example, is the set of 32-bit integers. A value for a message field can for instance be:

– A static value (such as a magic number).

– A value that depends on other field(s) in the same message, called an *intra-message* dependency (such as a checksum).

– A value that depends on other field(s) in a different message, called an *inter-message* dependency (such as a TCP acknowledgement number).

– A value depending on the environment, called an *environmental* dependency (such as a timestamp).

– A (semi-)random value (such as a TCP sequence number).

Protocol message formats can include optional or alternative fields, and repetitions. Moreover, a field can be composed of sub-fields. The main challenge in message format reverse engineering is to find the boundaries and (complex) relations between message fields and sub-fields. Some protocols make use of *delimiters* to denote field boundaries. Delimiters are protocol constants that are used to mark the boundary of variable-length fields. Other protocols use *keywords* to denote the beginning of a new field (and hence the field boundary). Besides denoting field boundaries, keywords often contain semantic information about the contents of a field, or its relation with other fields (e.g. with a length field that indicates the length of another field).

Three types of relations between fields have been distinguished in literature. The common *sequential* relation captures the ordering between two adjacent fields in the message. The *hierarchical* relation reflects the fact that a field can be further divided into multiple sub-fields. Finally, the *parallel* relation reflects the fact that the positions of two or more fields are interchangeable in the message format.

The relations between message fields can be captured by a *message field tree*, in which each node represents a field in the message. Here, a child node represents a subfield (and thus sub-range) of its parent. The internal nodes of the tree represent *complex* [150] or *hierarchical* [97, 29] fields (these notions are synonymous), and the leaf nodes represent the smallest hierarchical units in the message. Complex fields can exhibit intra-field or inter-field dependencies if a child refers to another part of the same field or a different field, respectively. Moreover, each node contains an attribute list, which captures properties of the field, such as its range.

A *message type* is a label that is given to a group of similar messages. Message types are used in the *protocol state machine* to define when it should transition to a different state. In some cases, message types are derived from the message field tree. In other cases, they are an abstraction of similar messages. Often, such an abstraction can be obtained by performing a clustering algorithm (or other machine learning techniques) on a set of messages.

A *session* describes a dialogue between two parties. It is a series of messages that go in alternating directions of communication to accomplish a specific task. The structure of a session is determined by the protocol state machine, which specifies the valid messages and expected replies at each position in a session.

Protocol inference consists of two tasks.

Message format reverse engineering is the process of automatically extracting specifications for valid messages in a protocol. It consists of reverse engineering the message format for one or more messages,

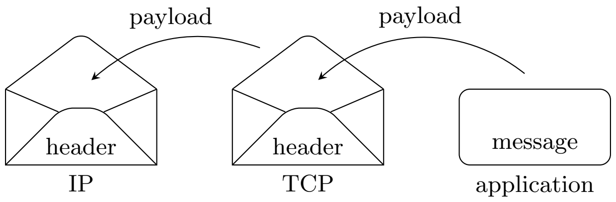


Figure 7.1: Encapsulation of application protocol messages.

and optionally of aggregating this information to determine a more general specification for the messages.

Protocol state machine inference is the process of inferring the input/ output behaviour of the protocol by observing protocol sessions as a sequence of messages. This often requires that message format reverse engineering and/ or machine learning techniques are used beforehand to create a finite set of message types.

In this chapter, we focus on the first task.

We are particularly interested in the *Internet Protocol Suite*. Therefore, we give a brief introduction to some of its relevant architectural principles here. The Internet Protocol Suite is a collection of protocols that specify how data should be exchanged on the Internet and similar networks. Its functionality is divided into layers of abstraction. Of particular interest to us are the *application layer*, which contains protocols for message exchange between applications. Some notable examples of protocols in the application layer are the *Hypertext Transfer Protocol* (HTTP), *Simple Mail Transfer Protocol* (SMTP) and *Secure Shell* (SSH) protocol.

Below the application layer are the *transport layer* and the *internet layer*, which contain the *Transmission Control Protocol* (TCP) and Internet Protocol (IP) respectively. TCP/IP provides reliable, ordered, and error-checked delivery of application layer protocol messages by *encapsulating* them in a series of packets. Figure 7.1 illustrates how encapsulation works, by making the distinction between *headers* and *payload.*

Table 7.1: Typical information sources for protocol inference techniques.

	passive	active
black-box	observation of network traffic	testing
white-box	observation of execution traces	fuzzing or symbolic execution

7.3 Classification of Protocol Inference Techniques

Protocol inference techniques can be classified based on the information that they use (this is true for those that reverse engineering the message format, as well as for those that infer the protocol state machine). In this section we give such a classification, and we highlight the strengths and weaknesses of the different approaches. Typical sources of information for the different approaches discussed in this section are shown in Table 7.1.

Passive vs active learning

One important classification of techniques for protocol inference is between *passive learning* and *active learning* approaches. In passive learning one observes (sequences of) messages to learn their characteristics, whilst in active learning approaches one actively takes part in the communication, either by playing the role of one of the involved parties, or by mutating the observed messages.

Each approach has its strengths with regard to their application in security. Passive learning can provide statistical information about the normal behaviour of a protocol, which active learning cannot. Such information can provide a basis for anomaly detection, which can be used for intrusion detection.

Passive learning approaches have the advantage that they do not require access to an implementation of the protocol. Instead, they only have to be able to observe its behaviour. A message format or state machine that is generated by a passive approach is, however, limited by the diversity of information seen. If certain variable fields never take more than one value, for example, it is impossible to identify those fields as variable fields. Similarly, it is impossible to infer transitions in the state machine for messages that never occur.

Active learning approaches, on the other hand, have the advantage that they can try messages for which the result is unknown. This way, they may try out strange corner cases that never occur in practice, and which would never be observed in passive learning. This can be useful to obtain a unique fingerprint for an implementation, or look for security flaws that arise in corner cases.

Black-box vs white-box

A second important classification of techniques for protocol inference is between those that consider the protocol implementation a *black box* and those that use it as a *white box.*

Black-box techniques only look at the messages that the implementation receives and produces. For communication protocols, they can obtain this information by looking at the network traffic. White-box techniques also observe some internals of the protocol implementation. By analysing an implementation while it processes an incoming message, for example, one may learn that different messages result in different execution paths; or by tracing the message through the execution, one may learn how the message is inspected or chopped up, which reveals information about its format.

An obvious advantage of a black-box technique is that it does not require access to the implementation. An advantage of white-box approaches is that the additional information may provide extra insight into the protocol. Specifically, an implementation may reveal semantic information about how it processes and operates on these messages. As a result, techniques that (are able to) consider the implementation as a white box are typically more accurate and provide richer information about the message format [30].

In the context of communication protocols (passive) black-box approaches are sometimes referred to as being *network based* (because they only look at network traffic), while (passive or active) white-box approaches are considered *host based* (because they look at the implementation on the host machine). We find this classification is confusing, however, because it does not clearly communicate the (orthogonal) distinction between passive and active learning. Therefore, we use the more general classification presented in this paragraph.

7.4 Reverse Engineering Message Formats

The first task in protocol inference is to obtain a specification of (some or all) valid input messages. For some protocols such specifications are publicly available, complete and up-to-date. For others, such specifications do not exist. The traditional way of obtaining protocol message format specifications is a notoriously laborious task that involves a significant amount of manual analysis [43]. Even when executable code is available, the task is complicated and error-prone. Manually reverse engineering the SMB protocol, for example, took 12 years in the open source SAMBA project [138]. Hence, automatic reverse engineering of message formats is an invaluable step in the process of protocol inference. Not only is this task (to some extent) a prerequisite for protocol state machine learning, but also does it have applications in security on its own.

Reverse engineering of message formats is a challenging task for a number of reasons [97]:

1. A message may contain a large number of fields.
2. Individual fields may not be static and may have a varying size.
3. There may exist sequential, parallel or hierarchical relationships and dependencies between fields.

To address these challenges, several tools and techniques have been proposed. These can be distinguished by their scope and analysis type.

Analysing individual messages In the simplest case, messages are analysed in isolation. Here, the aim is to reverse engineer the individual message formats of a protocol by identifying its fields.

Analysing multiple messages Commonly, different messages of a particular type do not contain the same fields in the same order. In such a case, the previous notion of message format reverse engineering falls short. A more general approach to the problem involves the analysis of a set of messages of a particular type. This produces a message format specification that can include alternative structures for different message types.

Related to the analysis scope is the type of the analysis that systems do. As stated above, there may exist complex relationships and dependencies between fields of a message. Most early systems, which analyse individual messages, consider a message format to be "flat". While this might be true for some simple or artificial protocols, many (real-world) protocols have sequential, parallel or hierarchical relationships between message fields [150]. To reflect this, systems that analyse multiple messages often represent them in a message field tree.

In the following subsections, we give a general description of the different tools and techniques that have been proposed for the task of automatically reverse engineering protocol message formats. First, we describe the passive, black-box approaches that focus on inferring message formats from network traces only. Then, we describe the approaches that leverage the availability of an implementation of the protocol. We conclude this section with an overview of work that is closely related to message format inference. An overview of the tools that we describe in this section, and a summary of their contributions can be found in Table 7.2.

Table 7.2: Overview of tools for protocol message format reverse engineering.

Tool	Novelty	Reference
PI	Application of sequence alignment to network traces for automatically finding message field boundaries.	[17]
ScriptGen	Extension of sequence alignment to region analysis for raising the training data to a higher level of abstraction. Application in a protocol emulator for honeypots.	[95, 94]
RolePlayer	Application of sequence alignment on a small set of cleverly constructed training examples for protocol replay.	[44]
Discoverer	Protocol independent solution for automatically reverse engineering the protocol message formats of an application from its network trace through clustering.	[43]

TaintCheck	Introduction of dynamic taint analysis for application fingerprinting.	[110]
Polyglot	Application of dynamic taint analysis for inferring message formats.	[30]
AutoFormat	Use of a context-aware execution monitor for inferring complex relationships and dependencies among message fields in a message field tree.	[97]
	Aggregation of individual message formats to infer a more general one.	[150]
Tupni	Extension of [150] to make message format reverse engineering more applicable to security applications.	[45]
ReFormat	Message format inference from encrypted data by performing data lifetime analysis.	[149]
Dispatcher	Infer encrypted message formats for both directions of communication, and annotate the message field tree with field semantics.	[29]
Veritas	Usage of the frequency distribution of n-grams in network traces for keyword detection.	[148]
Netzob	Combination of sequence alignment and clustering algorithms to group similar messages and associate message types.	[24]
ProDecoder	Accurate network-based message format inference of asynchronous and sampled protocol data.	[147]
Prisma	Define similarity between messages by embedding them in a vector space.	[85]
Pulsar	Improving inferred message formats with fuzzing.	[59]
AutoGram	Inference of context-gree grammars for message formats.	[74]

7.4.1 Inference from Network Traces

Early approaches for reverse engineering message formats use machine learning techniques to find patterns in network traces, and use these patterns to infer a message format. They can therefore be considered as primarily black-box, passive learning techniques. In this section, we give an overview of the advancements in this area, and the tools that have been created over the years.

Using sequence alignment for finding similarities

The first preliminary technique for protocol message format reverse engineering was introduced by the Protocol Informatics project [17]. Inspired by bioinformatics, it uses *sequence alignment* to find similarities in two or more messages of the protocol. Sequence alignment is a way of arranging two sequences to identify regions of similarity [107]. In bioinformatics it is used to understand the relationship between two sequences of genetic information, such as DNA or amino acids. For protocol analysis, the concept is similar. The goal is to compare a message to a database of messages belonging to a specific protocol. This allows one to determine its type and the location and size of the fields in each individual message. Sequence alignment is particularly effective on (sets of) messages in which dynamic fields have variable lengths [24]. Beddoe presents preliminary results in learning message formats for HTTP [17].

The sequence alignment algorithm of Needleman and Wunsch [107] was later implemented in the Netzob tool for partitioning messages in simple (i.e. non-complex) fields [24, 25]. The tool uses this technique in conjunction with a clustering algorithm to group together similar messages. This is used as a preprocessing step for inferring the domain of individual message fields, and (consecutively) the message type.

Detecting fields with region analysis

Several tool extend the work done in the Protocol Informatics project by using previously seen messages to heuristically detect some specific fields (such as network addresses, lengths, and cookies).

One such a tool ScriptGen [95, 94], which uses sequence alignment as a building block for a more complex algorithm, called *region analysis*. Region analysis consists of two steps. By looking at aligned sequences of bytes, it first computes for each aligned byte:

– its most frequent *type* of data (binary, text or zero-value),

– its most frequent *value*,

– the *variability* of the values, and

– the presence of *gaps* in aligned sequences.

Then, fields are identified on this basis of sequences of bytes that have some similar characteristics. By taking advantage of the statistical diversity of a large number of training messages, region analysis can be used to rebuild a partial notion of the semantics in a message format.

The independently developed RolePlayer tool [44] uses a similar technique as ScriptGen. Instead of using a large number of training samples, however, RolePlayer uses a small set of cleverly constructed samples to train the sequence alignment algorithm.

Towards a more complete approach for analysing network traffic

In 2007, Cui et al. introduced Discoverer [43]. The goal of this tool is to automatically reverse engineer message formats by analysing sequences of network packets. The idea of Discoverer is to cluster messages with the same record patten together and learn multiple message format specifications for a single protocol. This is achieved in three phases: *tokenization*, *clustering* and *merging*. In the following paragraphs we describe these phases in more detail. An overview of Discoverer's system architecture can be found in Figure 7.2.

First, consecutive network packets are reassembled in messages determined by the direction of communication. Then, the message is split into a sequence of *tokens*. A token is a sequence of consecutive bytes likely to belong to the same message field. Two types of tokens are distinguished: *text* and *binary*. Text segments are identified by comparing a sequence of bytes with the ASCII values of printable characters. A set of predefined

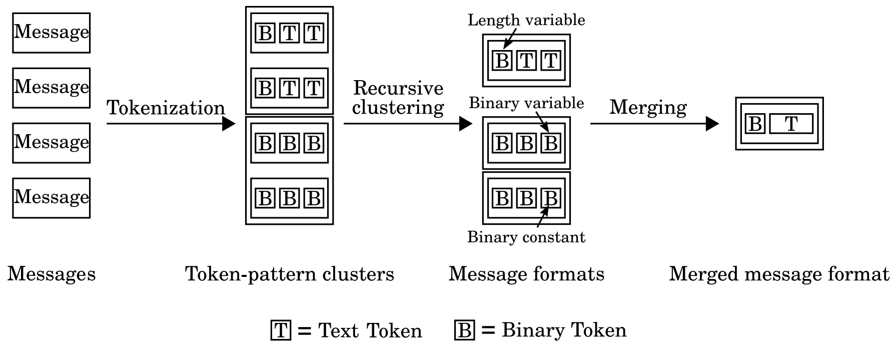


Figure 7.2: Overview of DISCOVERER's architecture [43, Figure 1]

delimiters is used to divide a text segment into tokens. The authors argue that identifying binary field boundaries is very hard. Therefore, they consider each binary byte to be a token in its own right.

Messages are clustered based on their token pattern. However, since messages with the same token pattern do not necessarily have the same format, clusters of messages are further divided so that each message in a cluster has the same format. Then, constant and variable length tokens are identified by comparing them against their counterparts in another message that has the same format. Three field semantics are inferred:

length the size of a field,

offset the byte offset of a field from a certain point (such as the start of the message),

cookie session-specific data that appears in messages from both sides of the application session (such as a session ID).

The key observation behind the merging phase is that sequence alignment can be used to identify similar message formats across different clusters. This is because we can leverage the token properties (text or binary, variable or fixed length) and semantics (length, offset and cookie) inferred in the previous phases. The authors have demonstrated that DISCOVERER can partially infer message formats for three application protocols: SMB, RPC, and HTTP.

Discoverer has three major limitations. First, it assumes the existence of a (set of) predefined delimiter(s) for dividing a text segment into tokens. However, protocols may not use delimiters and even if they do, these delimiters might not be available to the public. Second, it does not work for asynchronous application protocols, or (synchronous) protocols that are sampled. This is because Discoverer assembles raw packets into messages by grouping each sequence of consecutive packets that flow in one direction. This way of grouping packets is inappropriate, because two parties might send packets to each other at the same time. Moreover, a raw packet trace might be sampled, which severely reduces Discoverer's approach for the same reason. Third, the tool assumes that the first constant number of bytes of a session describe the complete message format. Whilst this is the case for the application protocols that were used in the experiments, this assumption does not hold all application protocols. The SMTP protocol, for example, indicates the end of the mail data by sending a line containing only a ".". This expression is part of the message format, while the content of the message itself can be of any (variable) length.

Using frequency distributions and n-grams

The limitations of Discoverer were addressed by Wang et al. in their Veritas and ProDecoder tools [148, 147]. These are fully automatic network-based tools for learning message formats that do not assume any prior knowledge of a protocol specification (such as delimiters). Similar to Discoverer, they are applicable to both text and binary protocols.

The key insight behind these tools is that *n-grams* in protocol messages exhibit a highly skewed frequency distribution that can be used for inferring its message format. An n-gram is a contiguous subsequence of n elements in a given sequence of at least n elements. In the case of Veritas and ProDecoder, an n-gram is a sequence of n bytes in a protocol message.

Veritas and ProDecoder consists of the following modules:

n-gram generation The input to this module is a set of raw network traces that are of the same protocol. These packets do not necessarily have to consist of (complete) protocol messages. Therefore, the tools are applicable to asynchronous protocols and sampled protocol data as well. In this module the raw packets are decomposed in subsequences of n contiguous bytes and the count for each such n-gram are stored. For example, if the parameter $n = 4$ then the n-grams from the message `MAIL FROM` are `MAIL`, `AIL_`, `IL_F`, `L_FR`, `_FRO` and `FROM`.

Keyword unit selection In Veritas, a Kolmogorov-Smirnov test filter is used to identify the frequent n-grams from the distribution created in the previous module. These frequent subsequences are called *keyword units* (Keyword units are called *message units* in [148]). The set of aforementioned n-grams, for example, can be discovered as keyword units, because they are encountered regularly. ProDecoder skips this module.

Keyword identification This module uses the keyword units collected in the previous module to infer keywords. Keywords are identified be searching for keyword units that often occur together. The aforementioned keyword units can be used to reconstruct the keyword `MAIL FROM`, because they occur together often. A message can have multiple keywords.

Message clustering This module clusters messages based on their keywords using standard machine learning techniques. Veritas uses the Jaccard index to calculate the similarity between messages. ProDecoder uses a standard hierarchical clustering method. The clusters are validated by using a metric from information theory known as the *information bottleneck method* [136]. This method captures the relevant information in a message with respect to the other messages by compressing the data. This enables ProDecoder to cluster messages based on their semantics, and distinguish among similar keywords belonging to different protocol messages.

Sequence alignment Similar to Discoverer, this module uses sequence alignment on the messages in each cluster to find the common byte sequences among them. These sequences represent the stable parts of the protocol messages, and can therefore be used to represent the message format. Veritas does not perform this final step. Instead, it uses the message format and the raw network traces to infer the protocol state machine.

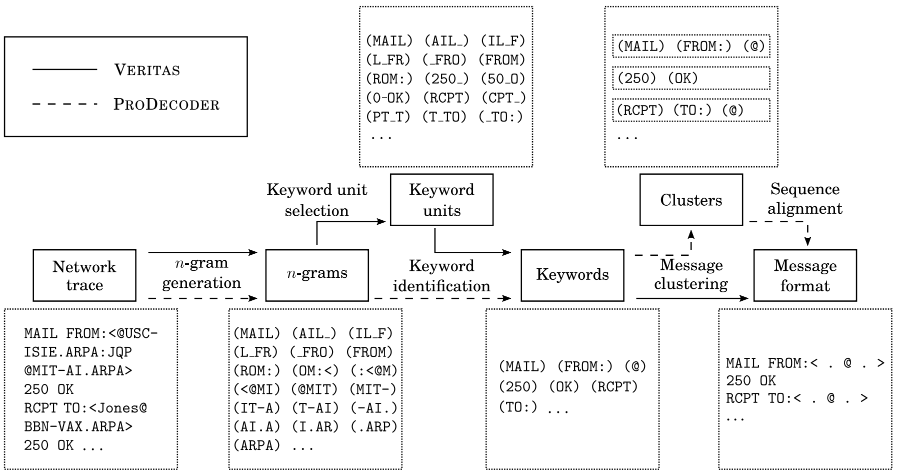

VERITAS
PRODECODER
Network trace
n-gram generation
n-grams
Keyword unit selection
Keyword units
Keyword identification
Keywords
Message clustering
Clusters
Sequence alignment
Message format
MAIL FROM:<@USC-ISIE.ARPA:JQP @MIT-AI.ARPA> 250 OK RCPT TO:<Jones@ BBN-VAX.ARPA> 250 OK ...
(MAIL) (AIL_) (IL_F) (L_FR) (_FRO) (FROM) (ROM:) (OM:<) (:<@M) (<@MI) (@MIT) (MIT-) (IT-A) (T-AI) (-AI.) (AI.A) (I.AR) (.ARP) (ARPA) ...
(MAIL) (AIL_) (IL_F) (L_FR) (_FRO) (FROM) (ROM:) (250_) (50_0) (0-OK) (RCPT) (CPT_) (PT_T) (T_TO) (_TO:) ...
(MAIL) (FROM:) (@) (250) (OK) (RCPT) (TO:) ...
(MAIL) (FROM:) (@)
(250) (OK)
(RCPT) (TO:) (@)
...
MAIL FROM:< . @ . > 250 OK RCPT TO:< . @ . > ...

Figure 7.3: Overview of Veritas' and ProDecoder's architectures (from [148, Figure 1] and [147, Figure 2]).

An overview of Veritas' and ProDecoder's architectures is shown in Figure 7.3. The authors have implemented and evaluated Veritas to infer messages format specifications for SMTP and two binary peer-to-peer protocols. ProDecoder is evaluated on SMB and SMTP network traces. The experimental results show that both tools accurately parse the application protocols.

Using more advanced methods

The work of Wang et al. was extended by Krueger et al. in their Prisma tool [85]. Instead of using n-grams for finding similarities between different messages, the authors use a more elaborate method. To find common structures in the data, they first define a similarity measure between messages. This is done by embedding the messages in special vector spaces which are reduced via statistical tests to focus on discriminative features. In contrast to previous tools, the model constructed by Prisma can not only analyze but also simulate messages.

7.4.2 Using the Protocol Implementation

While the passive, black-box approaches presented in the previous section have the advantage that they do not require access to an implementation of the protocol, they have limitations.

Trace dependency The message format that is generated by a passive approach is limited by diversity of messages seen. If certain messages never occur, it is impossible for a tool to infer their message formats. Similarly, if certain variable fields never take more than one value, it is impossible to identify those fields as variable fields [43].

Semantics It was observed that the lack of protocol semantics in network traces fundamentally limits the precision of the extracted message formats [45, 30].

Encryption Black-box approaches are easily hampered by encryption, because they do not take the end-points of the communication dialog in account [97].

Instead of solely looking at network traces, most modern approaches for message format reverse engineering leverage the availability of a (binary) implementation of the protocol. Compared to the syntactic information in network traces, implementation binaries contain semantic information about how the protocol operates on the input messages.

Given an implementation of a protocol, the goal is to find the set of messages that the implementation accepts. This is an undecidable problem in general. Instead, one can monitor how an implementation (in the form of an application binary) processes the input (messages) that it receives, instead of analysing the messages or the implementation in isolation.

Indeed, most modern approaches for message format reverse engineering can be considered white-box. Some techniques use a passive learning approach, and analyse the implementation while it processes network traces. Other approaches use an active learning approach, and try to fuzz the implementation.

Both of these approaches typically consist of two phases. In the first phase it observes an implementation (in the form of a program binary) as it processes a message. The intuition behind this approach is that knowing which part of a program processes which part of an input can reveal the structure of the input as well as the structure of the program. This component of the system, typically called the *execution monitor*, takes as input a program binary and an input (message), and applies *dynamic taint analysis* to monitor how the program processes the data by instrumenting the code [110]. Recently, the potential of dynamic taint analysis for message format reverse engineering has increased with the introduction of general purpose taint tracking systems (for example for JVM [18]).

As the protocol implementation executes, the propagation of tainted input is monitored by observing memory buffers and the operations that are applied to them. The result of this is often referred to as an *execution trace* [30] (in the remainder of this section we will therefore use this term to refer to the result of dynamic taint analysis, even if the paper we discuss uses a different term). An execution trace typically consists of (some) of the following information [18]:

Method calls are logged with the taint information for all argument values.

Method returns are logged with the taint information for its return value.

Arithmetic operations are logged with the taint information for its operands.

Memory access is logged with its access type (read or write) and the taint information of the value that was read/written.

Array instructions are logged with its access type (read or write) and the taint information of the value that was read/written.

In the second phase the execution trace is analysed. For this task, a number of techniques have been proposed, which we will discuss in the remainder of this section.

Using dynamic taint analysis

Dynamic taint analysis refers to monitoring how a program processes its input data by instrumenting the code [110]. Recently, the potential of dynamic taint analysis for message format reverse engineering has increased with the introduction of general purpose taint tracking systems (for example for JVM [18]).

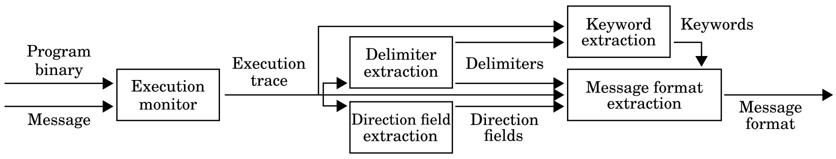


Figure 7.4: Overview of POLYGLOT's architecture [30, Figure 1]

One of the first tools that uses dynamic taint analysis for learning protocol message formats was introduced by Caballero et al. [30]. The objective of their tool, called POLYGLOT, is to find the field boundaries in individual input messages to form the basis for the message format for these messages. POLYGLOT is based on TAINTCHECK [110], a tool that used dynamic taint analysis on network traces from untrusted sources for automatically generating signatures (fingerprints) for malware attacks. TAINTCHECK monitors how each byte of a network trace is used by the implementation at the processor-instruction level.

POLYGLOT uses the execution trace for reverse engineering of its message format. The execution trace is analysed by four modules that locate the field boundaries and keywords. In the following paragraphs, we will describe these modules in more detail. An overview of POLYGLOT's system architecture can be found in Figure 7.4.

First, boundaries of variable-length fields are located by the *direction field* and *delimiter extraction*[1] modules. A direction field stores information about the location of another (target) field in the message, such as its length. The intuition behind detection direction fields is that they are *used* during execution of a trace (to increment a memory pointer to the tainted data, for example). Delimiters are elements that are used to mark the boundary of variable-length fields. To find these delimiters, Caballero et al. look for tokens that are compared against consecutive positions in the stream of data [30].

The *keyword extraction* module takes as input the delimiters and the execution trace and outputs the protocol keywords. The problem is to extract the subset of keywords that are supported by the implementation

[1]called *separator extraction* in [30].

and present in the network data. Keywords are known a priori by the protocol implementation. Hence, when a message arrives, the implementation compares its keywords against the received data. The intuition is that one can locate these keywords by checking for comparisons between tainted and untainted data, since the network input will be tainted and for example constants hardcoded in the implementation will be untainted.

Finally, the *message format extraction* module combines all previous information to generate the message format. The authors have evaluated Polyglot over five different protocols: DNS, HTTP, IRC, SMB and IRQ. Compared to manually crafted reference specifications, the reverse engineered message formats show minimal differences. These differences are mostly due to different implementations handling fields in different ways.

Representing complex relationships and dependencies

One drawback of Polyglot is that it takes a "flat" view of a message format. This problem was addressed by Lin et al., who argue that such a flat view cannot represent complex relationships and dependencies among the fields [97]. The authors observe that to reverse engineer message formats more accurately and thoroughly, in addition to the extracting fields, it is important to expose cross-field relations and reveal the hierarchical structure of the message formats. The focus of their work is the determination of three important types of relations between message fields:

AutoFormat [97], is one of the first tools that aims to automatically infer such a message field tree. It distinguishes fields based on the observation that different fields in the same message are typically handled in different execution contexts. By monitoring program execution, a *context-aware* execution monitor first collects execution context information for every byte of the network trace. Then, the message *field identifier* identifies fields based on this context information. If two successive bytes share the same context information, then they are clustered. Finally, the field identifier constructs a message field tree, by identifying parallel and sequential fields. Parallel fields are discovered by comparing the execution context. Sequential fields are discovered by recursively traversing the non-parallel fields.

Experimental results show that AutoFormat achieves high accuracy in message field identification and message format reconstruction for seven real-world protocols, including two text-based protocols (HTTP and SIP), three binary-based protocols (DHCP, RIP, and OSPF), one hybrid protocol (SMB) and an unknown malware protocol.

Analysing different types of messages

Where Polyglot and AutoFormat infer message formats for *individual* messages, the tool introduced by Wondracek et al. can aggregate these individual message formats to infer a more general one [150]. Given a set of input messages, the tool can be used to parse different types of messages of the protocol. This allows them to extract some additional semantics about the message format, such as identifying optional fields. The authors apply their techniques to a set of real-world server applications that implement the HTTP, DNS, SMTP, SMB and NFS protocols.

It was observed, however, that messages may have features that hamper the applicability of the work of Wondracek et. al. First, messages may include data records of *arbitrary* length, and second, there may exist arbitrary cross-field and cross-message references (such as checksums or sequence numbers), which cannot be captured by the proposed semantics and structure.

In response, Cui et al. propose Tupni, a tool that attempts to solve these problems [45]. The system takes as input one or more messages of the unknown format, and a program that can process these inputs. Unlike the previous tools, Tupni can identify arbitrary sequences of data records by analysing loops in the execution trace of the input message(s). The tool recognises loops by analysing the instructions that a message invokes. It considers messages as belonging to the same type if the loop iterations execute mostly the same instructions.

The authors demonstrate that Tupni can be used to reverse engineer message formats for DNS, RPC, TFTP, HTTP and FTP. Apart from network protocols, the tool was also applied to infer file formats.

After extracting information about the structure of individual messages, both Tupni and the tool by Wondracek et al. can aggregate this structure over different message types: if multiple messages of an unknown message format are available, the tools can perform their analysis on each of the resulting execution traces, and combine the individual results into a single message format specification.

Handling encryption

Despite their effort, a common limitation of all aforementioned tools is that they cannot infer the message format when the network traffic is encrypted. This problem is addressed by Wang et al., who introduce ReFormat, a tool that aims to derive the message format even when the message is encrypted [149]. Their approach is based on the observation that encrypted network traffic typically goes through two processing phases: first decryption and then normal protocol processing. Based on this observation, the authors use a different source of input to solve the problem: it is not the network traffic whose message format they try to infer, but the *memory buffer* that contains the decrypted message at run-time.

ReFormat can accurately identify the buffers that contain the decrypted message, because the function calls and instructions for decrypting an encrypted message are significantly different from those used for processing an unencrypted message. After separating the decryption phase from the processing phase, the tool performs so called *data lifetime analysis* to pinpoint the memory buffers that contain the decrypted message (Called *buffer deconstruction* in [29]).

Experimental analysis shows that ReFormat can accurately identify message formats for four protocols that encrypt their network communications: HTTPS, IRC, MIME and an unknown malware protocol.

Understanding field semantics

The works of Lin et al. [97] and Wang et al. [149] were extended by Caballero et al., who introduce Dispatcher [29]. In this work, they address three problems.

First, they argue that it helps to understand *what* a message does when trying to infer its message format. They address this problem by annotating nodes in the message field tree with *field semantics*. Second, they extract the message format for *both* directions of communication. This choice is motivated by the observation that security analysts frequently have to rewrite protocol messages *sent* by an application. Third, they extend the approach of [149] in coping with encrypted messages.

Together, they apply these contributions to extract message formats for unknown, encrypted, malware protocols. Dispatcher was used to analyse and infiltrate the previously undocumented command and control protocol of MegaD, a spam botnet. In the following paragraphs, we will describe the three problems that Dispatcher addresses in more detail.

Field semantics describe the type of data that a field contains, such as the message length, a host name, or a time stamp. The intuition behind type-inference-based techniques is that the function and system calls used by programs contain rich semantic information about their goal, arguments and output. This information is publicly available in so-called *prototypes*. Dispatcher maintains a set of prototypes for commonly used functions and instructions. For inferring the field semantics of *received* messages, it uses dynamic taint analysis to monitor if a sequence of bytes from a received message is used in the arguments of some function calls or instructions in the execution trace. That sequence of bytes can then be associated with the semantics defined in the prototype.

Similarly, for *sent* messages, Dispatcher taints the output of interesting functions and instructions with unique identifiers. For each tainted sequence in the network trace, it looks up the identifier in the set of known prototypes. Most data in sent messages does not come from the tainted network traces. Instead, Dispatcher uses the intuition that programs store fields in a memory buffer and construct messages by combining these buffers.

To deal with encrypted data, Dispatcher uses the same technique as ReFormat (it identifies the memory buffers holding the unencrypted data). It extends this approach to analyse the data sent as well as data received: for incoming data it identifies the memory buffers holding the incoming data *after* it is *de*crypted (just as ReFormat does). For outgoing data it also identifies the memory buffers that hold the outgoing data *before* it is *en*crypted.

Once these memory buffers have been identified, the discussed message format reverse engineering techniques can be applied on these buffers instead of on the network traces. Here another extensions to ReFormat [149] is introduced. ReFormat can only handle applications where there exists a single boundary between the decryption and processing phase. Caballero et al. observe, however, that multiple such boundaries exist for the MegaD botnet protocol [29]. Hence, they extend the tool to identify every instance of the decryption phase.

Using fuzzing

The Prisma tool, introduced in the previous section, was improved by Gascon et al. in their Pulsar tool. Pulsar proceeds by observing the network traffic of an unknown protocol and inferring a generative model for message formats. In contrast to previous approaches, this model enables effectively exploring the protocol state space during fuzzing and directing the analysis to states which are particularly suitable for fuzz testing. This guided fuzzing allows for uncovering vulnerabilities deep inside the protocol implementation. Moreover, by being part of the communication, Pulsar can increase the coverage of the protocol state space, resulting in less but more effective testing iterations.

Learning context-free grammars

Höschele et al. introduce the AutoGram tool, which can be used to infer a *context-free grammar* for valid input messages [74]. Such a grammar is a set of rewriting rules that can be used to both recognize (parse) valid input messages, and generate (syntactically valid) unseen ones. In addition, the inferred grammars are natural to read (for humans), because the authors have implemented simple heuristics to label the rewrite rules with (semantically) relevant names. The authors have primarily used their tool for the reverse engineering of data formats (URL, CSV, INI and JSON), but its relevance for communication protocol messages is evident.

AutoGram constructs a grammar by aggregating multiple input messages. As such, it can be seen as a continuation of the line of work initiated by Wondracek et al. [150]. For each message, the tool constructs an *interval tree*. Each layer of this tree partitions a message into fragments (intervals) based on information in the execution trace. Hence, each node in the tree corresponds with a fragment and the set of operations (i.e. method calls, arithmetic, return values ...) that were applied to that fragment. The rewrite rules are generated from a set of interval trees by combining the fragments in nodes whose operations are compatible.

Since AutoGram works with Java bytecode it does not have to resort to random names for the derived rewrite rules. Instead, it uses simple heuristics to extract a readable (and semantically relevant) name from the execution trace. This makes the grammars that the tool generates readily applicable for security analysis, for example.

7.5 Applications

In the following paragraphs we give an overview of the applications of message format reverse engineering in security.

Protocol emulation or replay Several studies have performed some level of message format reverse engineering with a specific purpose in mind: *protocol emulation* [95, 94] or *protocol replay* [44, 109]. Often used in the context of malware, a protocol emulator or replayer aims to mimic an existing system in order to learn more about (possible) attackers of a system [95]. Constructing such a tool manually is a tedious and sometimes impossible task, especially for protocols for which no documentation exists [95].

ScriptGen and RolePlayer were developed to mimic an existing system while interacting with a host on the network. Besides (partially) inferring the message format, this involves learning (partially) the protocol's state machine. With regard to a protocol's message format, the goal of both tools is to identify interesting message fields to rewrite. ScriptGen was applied in *honeypots*. A honeypot is a resource that is expected to get attacked, with the aim of providing information about these attacks and the malicious payload that they deliver. Often, an attacker delivers this malicious payload only after a number of messages. The goal of ScriptGen is to automatically carry on a valid conversation for as long as needed for the attacker to deliver its malicious payload. In [94], the authors show that this is feasible. RolePlayer was used to replay both the client and server sides for a variety of network protocols, including NFS, FTP and SMB.

Anomaly detection Anomaly detection refers to trying to detect attacks by detecting changes in the communication traffic. For anomaly detection one will use passive learning techniques, during some learning phase in which there is only normal traffic, to then later detect deviations. Information about the normal traffic inferred can also be used to configure *intrusion detection systems* (IDS) and *intrusion prevention systems* (IPS). Modern IDSes and IPSes, such as FlowSifter [99], NetShield [96], GAPA [23] and binpac [113] use protocol message formats for this purpose. However, many application protocols in use are proprietary and have no publicly released specifications [87]. To parse traffic of these application protocols, protocol inference is required.

Fingerprinting When analysing different implementations of the same protocol, any differences between implementations, or between an implementation and the official specification, can point to security vulnerabilities. Such differences can also be used for *fingerprinting* a particular implementation of a protocol. A *fingerprint* can be used to uniquely identify such an implementation. Of course, protocol inference can also be used to fingerprint traffic of different protocols.

Security analysis A detailed understanding of a particular protocol can be used for a *security analysis* of that protocol. This can be a manual analysis, but might also be tool-supported, e.g. using model checkers (for example to detect possibilities for deadlock and hence Denial-of-Service) or using dedicated tools for security protocol analysis such as ProVerif [88].

Fuzzing An understanding of a protocol can also be used as a basis for *fuzzing*, where one randomly generates traffic which deviates from normal protocol traffic in the hope of catching implementation mistakes in the handling of 'incorrect' or 'malformed' input messages or message sequences [111].

Malware analysis Protocol inference can be used to analyse (implementations of) the benign protocols (for instance when configuring an IDS or IPS), but it can also be used to analyse *malware*.

Tainting, for example, is a useful technique for malware analysis. If tainted data is used in ways that are defined as illegitimate, the tool can provide information about how the exploit happened and what it attempts to do. For example, if tainted input data is executed, by loading it into the instruction register of the CPU, this can be the sign of a classic buffer overflow attack where the attacker is executing his own shell code [37]. This information is useful for identifying vulnerabilities and for generating malware signatures (fingerprints). TaintCheck provides basic functionality for this, but is used for different purposes.

Other research in this area has focused on the automated reverse engineering of the protocol used by botnets [40]. A difference is that such protocols are typically unknown, whereas the protocols we want to protect will be known. Note that here protocol inference can be both an offensive and defensive technique: the aim can be to fix any of the security flaws found, or to exploit them. There is no fundamental distinction between the two, as the difference is often only a matter of context. Even offensive uses of protocol inference have their white-hat uses. For example, the same technique an attacker uses, say, to install a botnet, can then be used by a defender to analyse this botnet in order to try to take it down. Similarly, fingerprinting can be used as an offensive technique by an attacker when trying to detect the use of a particular implementation (namely, one with a known security vulnerability), but it can be used a defensive technique in trying to detect the presence of a botnet.

7.6 Conclusion

In this chapter, we have given an overview of research on reverse engineering protocol message formats. The goal of this work was to bridge the gap between the academic and the applied world in the field of protocol inference for security applications. The contributions of this chapter were threefold. First, we have given a common terminology for different aspects of communications protocols and a general classification of protocol inference techniques. Then, we have given a systematic overview of the techniques that have been proposed for reverse engineering message formats. Finally, we have given an overview of the applications of these techniques in security.

Chapter 8

Conclusion

This thesis addresses multiple problems in the area of model learning for software systems. In addressing these problems, I believe that my co-authors and I have made significant contributions to the field. It is my hope that these contributions make model learning more practically applicable for reverse engineering reactive software systems. In addition, I hope that the work in this thesis sparks many interesting new directions for research in the area.

In this conclusion, I give a brief summary of the lessons learned in this thesis and some directions for future work.

Metrics in model learning

Chapters 3 and 4 advocate the use of metrics in model learning. We have presented a general class of distance metrics on Mealy machines that may be used to formalize intuitive notions of quality, and we have used two such metrics to reduce the number of inputs required while learning. We conjecture that the utility of metrics in model learning increases as models become more complex – both qualitatively (i.e. that they help understanding the quality of an hypothesis better) and quantitatively (i.e. that they may lead to a reduction in the number of inputs). A possible direction for future work is to verify this, by applying our methods to more complex systems. Another related direction for future work is to devise new intuitive metrics that can be used in this setting, and see if they result in a reduction in inputs required for learning.

Bounding the distance between a hypothesis and the unknown target model during learning remains a challenging problem. Using our metrics, the quality of a hypothesis is hard to predict because of the high variance for different experimental runs. Different metrics might produce more encouraging results in this sense.

SMT solvers for model learning

In Chapter 5 we explore an approach to model learning that is based on using SMT solvers. We have provided encodings for DFAs, Mealy machines and register automata, and an implementation of these encodings in Z3.

We believe that this chapter might give rise to a broader direction of future work, since the approach has several advantages over traditional model learning algorithms:

1. it typically requires fewer queries,
2. it is more easily adaptable to other formalisms, and
3. it is applicable for both active and passive learning, and a combination of both.

Since we are not experts on SMT solvers, some effort is required to implement the encodings that we present more efficiently. A major direction for future work is to explore the application of this approach in a combination of active and passive learning more. Another major direction for future work is to encode formalisms for which no learning algorithms exist yet.

Fuzzing and model learning

In Chapter 6 we shown that conformance testing and mutation-based fuzzing are orthogonal and complementary approaches for finding counterexamples in model learning for software systems. A major direction for future research is to combine these methods in more interesting ways. We have already made some effort in this sense:

1. use fuzzing as a source of counterexamples *during* learning, and
2. use (intermediate) learning results to guide mutation-based fuzzing.

Another direction for future research is to apply fuzzing while learning more expressive formalisms, such as register automata.

Reverse engineering message formats

Chapter 7 presents an overview of the techniques for reverse engineering the message formats of communication protocols and other reactive software systems. In most cases, this is a prerequisite for model learning.

The goal of this work is to bridge the gap between the academic and the applied world in this area. We believe that this chapter gives a unified and detailed overview of the techniques that are available. It lacks an experimental evaluation of the tools that are available, however. Therefore, a possible direction for future work is to devise a set of benchmarks for this purpose, and to give an quantitative assessment of the performance of the tools that are available.

Bibliography

[1] F. Aarts. *Tomte: bridging the gap between active learning and real-world systems*. PhD thesis, Radboud University Nijmegen, 2014.

[2] F. Aarts, J. de Ruiter, and E. Poll. Formal models of bank cards for free. In *Proceedings ICTW*, pages 461–468. IEEE, 2013.

[3] F. Aarts, P. Fiterău-Broştean, H. Kuppens, and F. Vaandrager. Learning register automata with fresh value generation. In *Proceedings ICTAC*, volume 9399 of *LNCS*, pages 165–183. Springer, 2015.

[4] F. Aarts, P. Fiterău-Broştean, H. Kuppens, and F. Vaandrager. Learning register automata with fresh value generation. Technical report, Radboud University Nijmegen, 2016. `http://www.sws.cs.ru.nl/publications/papers/fvaan/TomteFresh/full.pdf`, accessed July 2017.

[5] F. Aarts, B. Jonsson, J. Uijen, and F. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, 46(1):1–41, 2015.

[6] F. Aarts, H. Kuppens, J. Tretmans, F. Vaandrager, and S. Verwer. Improving active Mealy machine learning for protocol conformance testing. *Machine Learning*, 96(1-2):189–224, 2014.

[7] F. Aarts, J. Schmaltz, and F. Vaandrager. Inference and abstraction of the biometric passport. In *Proceedings ISOLA*, volume 6415 of *LNCS*, pages 673–686. Springer, 2010.

[8] L. de Alfaro, M. Faella, and M. Stoelinga. Linear and branching system metrics. *Software Engineering*, 35(2):258–273, 2009.

[9] L. de Alfaro, T. A. Henzinger, and R. Majumdar. Discounting the future in systems theory. In *Proceedings ICALP*, volume 2719 of *LNCS*, pages 1022–1037. Springer, 2003.

[10] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proceedings POPL*, pages 4–16. ACM, 2002.

[11] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87 – 106, 1987.

[12] D. Angluin. Negative results for equivalence queries. *Machine Learning*, 5:121–150, 1990.

[13] D. Axe. Marines' first frontline stealth fighter lacks vital gear, 2012. `https://www.wired.com/2012/11/marines-jsf/`, date accessed: 2017-8-18.

[14] J. W. de Bakker and J. I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54(1–2):70 – 120, 1982.

[15] J. L. Balcázar, J. Díaz, R. Gavaldà, and O. Watanabe. Algorithms for learning finite automata from queries: A unified view. In *Advances in Algorithms, Languages, and Complexity*, pages 53–72. Springer, 1997.

[16] O. Bauer, J. Neubauer, B. Steffen, and F. Howar. Reusing system states by active learning algorithms. In *Proceedings EternalS*, volume 255 of *CCIS*, pages 61–78. Springer, 2012.

[17] M. A. Beddoe. Network protocol analysis using bioinformatics algorithms. Technical report, Baseline Research, 2004. `http://www.4tphi.net/~awalters/PI/PI.html`, retrieved 2016-03-10.

[18] J. Bell and G. Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *Proceedings OOPSLA*, pages 83–101. ACM, 2014.

[19] A. Bertolino, A. Calabrò, M. Merten, and B. Steffen. Never-stop learning: Continuous validation of learned models for evolving systems through monitoring. *ERCIM News*, 2012(88), 2012.

[20] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 21(6):592–597, 1972.

[21] A. Birkendorf, A. Böker, and H. U. Simon. Learning deterministic finite automata from smallest counterexamples. *Discrete Mathematics*, 13(4):465–491, 2000.

[22] F. Bonchi and D. Pous. Checking NFA equivalence with bisimulations up to congruence. In *Proceedings POPL*, pages 457–468. ACM, 2013.

[23] N. Borisov, D. Brumley, and H. Wang. Generic application-level protocol analyzer and its language. In *Proceedings NDSS*, 2007.

[24] G. Bossert and F. Guihéry. Security evaluation of communication protocols in common criteria. In *Proceedings ICCC*, 2012.

[25] G. Bossert, F. Guihéry, and G. Hiet. Netzob: un outil pour la rétro-conception de protocoles de communication. In *Proceedings SSTIC*, 2012.

[26] P. van den Bos. Enhancing active automata learning by a user log based metric. Master's thesis, Radboud University Nijmegen, 2015.

[27] L. Brandán Briones, E. Brinksma, and M. Stoelinga. A semantic framework for test coverage. In *Proceedings ATVA*, volume 4218 of *LNCS*, pages 399–414. Springer, 2006.

[28] M. Bruynooghe, H. Blockeel, B. Bogaerts, B. De Cat, S. De Pooter, J. Jansen, A. Labarre, J. Ramon, M. Denecker, and S. Verwer. Predicate logic as a modeling language: modeling and solving some machine learning and data mining problems with IDP3. *Theory and Practice of Logic Programming*, 15(6):783–817, 2015.

[29] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings CCS*, pages 621–634. ACM, 2009.

[30] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings CCS*, pages 317–329. ACM, 2007.

[31] R. C. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *Proceedings ICGI*, volume 862 of *LNCS*, pages 139–152. Springer, 1994.

[32] S. Cassel. *Learning Component Behavior from Tests: Theory and Algorithms for Automata with Data.* PhD thesis, Uppsala University, 2015.

[33] S. Cassel, F. Howar, and B. Jonsson. RALib: A LearnLib extension for inferring EFSMs. In *Proceedings DIFTS*, 2015.

[34] S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Active learning for extended finite state machines. *Formal Aspects of Computing*, 28(2):233–263, Apr 2016.

[35] P. Černỳ, T. Henzinger, and A. Radhakrishna. Simulation distances. In *Proceedings CONCUR*, volume 6269 of *LNCS*, pages 253–268. Springer, 2010.

[36] G. Chalupar, S. Peherstorfer, E. Poll, and J. de Ruiter. Automated reverse engineering using Lego®. In *Proceedings WOOT*, pages 1–10, 2014.

[37] S. Chen, X. U. Jun, Z. Kalbarczyk, and R. K. Iyer. Security vulnerabilities: From analysis to detection and masking techniques. *Proceedings IEEE*, 94(2):407–418, 2006.

[38] D. Chivilikhin, V. Ulyantsev, and A. Shalyto. Combining exact and metaheuristic techniques for learning extended finite-state machines from test scenarios and temporal properties. In *Proceedings ICMLA*, pages 350–355. IEEE, 2014.

[39] C. Y. Cho, D. Babic, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song. MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings USENIX Security*, pages 139–154, 2011.

[40] C. Y. Cho, D. Babić, E. C. R. Shin, and D. Song. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings CCS*, pages 426–439. ACM, 2010.

[41] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[42] F. Coste and J. Nicolas. Regular inference as a graph coloring problem. In *Proceedings ICML*, 1997.

[43] W. Cui, J. Kannan, and H. J. Wang. Discoverer: automatic protocol reverse engineering from network traces. In *Proceedings USENIX Security*, pages 199–212, 2007.

[44] W. Cui, V. Paxson, N. C. Weaver, and R. H. Katz. Protocol-independent adaptive replay of application dialog. In *Proceedings NDSS*, 2006.

[45] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: automatic reverse engineering of input formats. In *Proceedings CCS*, pages 391–402, New York, New York, USA, 2008. ACM.

[46] E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[47] R. Dorofeeva, K. El-Fakih, S. Maag, A. R. Cavalli, and N. Yevtushenko. FSM-based conformance testing methods: A survey annotated with experimental evaluation. *Information and Software Technology*, 52(12):1286–1297, 2010.

[48] A. Dovier, C. Piazza, and A. Policriti. A fast bisimulation algorithm. In *Proceedings CAV*, volume 2102 of *LNCS*, pages 79–90. Springer, 2001.

[49] M. Droste, W. Kuich, and H. Vogler. *Handbook of Weighted Automata*. Springer, 1st edition, 2009.

[50] A. Drusch. Fighter plane cost overruns detailed, 2014. `http://www.politico.com/story/2014/02/f-35-fighter-plane-costs-103579`, date accessed: 2017-8-18.

[51] F. Duchene, R. Groz, S. Rawat, and J. L. Richier. XSS vulnerability detection using model inference assisted evolutionary fuzzing. In *Proceedings ICST*, pages 815–817, 2012.

[52] J. Duchêne, C. Le Guernic, E. Alata, V. Nicomette, and M. Kaâniche. State of the art of network protocol reverse engineering tools. *Journal of Computer Virology and Hacking Techniques*, pages 1–16, 2017.

[53] P. Dupont. Incremental regular inference. In *Proceedings ICGI*, volume 1147 of *LNCS*, pages 222–237. Springer, 1996.

[54] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager. Combining model learning and model checking to analyze TCP implementations. In *Proceedings CAV*, volume 9780 of *LNCS*, pages 454–471. Springer, 2016.

[55] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager. Learning fragments of the TCP network protocol. In *Proceedings FMICS*, volume 8718 of *LNCS*, pages 78–93. Springer, 2014.

[56] P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg. Model learning and model checking of SSH implementations. In *Proceedings SPIN*, pages 142–151. ACM, 2017.

[57] C. C. Florêncio and S. Verwer. Regular inference as vertex coloring. *Theoretical Computer Science*, 558:18–34, 2014.

[58] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *Software Engineering*, 17(6):591–603, 1991.

[59] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *Proceedings SecureComm*, volume 164 of *LNICST*, pages 330–347. Springer, 2015.

[60] D. Giannakopoulou, Z. Rakamarić, and V. Raman. Symbolic learning of component interfaces. In *Proceedings SAS*, volume 7460 of *LNCS*, pages 248–264. Springer, 2012.

[61] G. Giantamidis and S. Tripakis. Learning Moore machines from input-output traces. In *Proceedings FM*, volume 9995 of *LNCS*, pages 291–309. Springer, 2016.

[62] A. Gill. *Introduction to the theory of finite-state machines.* McGraw-Hill, 1962.

[63] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[64] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[65] D. Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2(2):97–109, 1973.

[66] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[67] T. Henzinger. Quantitative reactive modeling and verification. *Computer Science - Research and Development*, 28(4):331–344, 2013.

[68] M. J. H. Heule and S. Verwer. Exact DFA identification using SAT solvers. In *Proceedings ICGI*, volume 6339 of *LNAI*, pages 66–79. Springer, 2010.

[69] M. J. H. Heule and S. Verwer. Software model synthesis using satisfiability solvers. *Empirical Software Engineering*, 18(4):825–856, 2013.

[70] R. M. Hierons and U. C. Türker. Incomplete distinguishing sequences for finite state machines. *The Computer Journal*, pages 1–25, 2015.

[71] C. de la Higuera. *Grammatical inference.* Cambridge University Press, 2010.

[72] J. E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, pages 189–196, 1971.

[73] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation.* Pearson, 1st edition, 1979.

[74] M. Höschele and A. Zeller. Mining input grammars from dynamic taints. In *Proceedings ASE*, pages 720–725. ACM, 2016.

[75] F. Howar, D. Giannakopoulou, and Z. Rakamarić. Hybrid learning: Interface generation through static, dynamic, and symbolic analysis. In *Proceedings ISSTA*, pages 268–279. ACM, 2013.

[76] F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In *Proceedings VMCAI*, volume 7148 of *LNCS*, pages 251–266. Springer, 2012.

[77] H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *Proceedings CAV*, volume 2725 of *LNCS*, pages 315–327. Springer, 2003.

[78] O. H. Ibarra and T. Jiang. Learning regular languages from counterexamples. *Computer and System Sciences*, 43(2):299–316, 1991.

[79] F. Ipate. Learning finite cover automata from queries. *Computer and System Sciences*, 78(1):221 – 244, 2012.

[80] M. N. Irfan, R. Groz, and C. Oriat. Optimising Angluin algorithm L* by minimising the number of membership queries to process counterexamples. In *Zulu Workshop*, page 134, 2010.

[81] M.-N. Irfan, R. Groz, and C. Oriat. Improving model inference of black box components having large input test set. In *Proceedings ICGI*, volume 21 of *JMLR W&CP*, pages 133–138, 2012.

[82] M. Isberner. *Foundations of Active Automata Learning: An Algorithmic Perspective*. PhD thesis, Technical University of Dortmund, 2015.

[83] M. Isberner, F. Howar, and B. Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *Proceedings RV*, volume 8734 of *LNCS*, pages 307–322. Springer, 2014.

[84] T. Knuutila. Re-describing an algorithm by Hopcroft. *Theoretical Computer Science*, 250(1-2):333–363, 2001.

[85] T. Krueger, H. Gascon, N. Krämer, and K. Rieck. Learning stateful models for network honeypots. In *Proceedings AISec*, pages 37–48. ACM, 2012.

[86] S. Kullback and R. A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[87] J. F. Kurose and K. W. Ross. *Computer networking: a top-down approach.* Pearson, 6th edition, 2013.

[88] R. Küsters and T. Truderung. Using ProVerif to analyze protocols with Diffie-Hellman exponentiation. In *Proceedings CSF*, pages 157–171. IEEE, 2009.

[89] K. J. Lang. Faster algorithms for finding minimal consistent DFAs. Technical report, NEC Research Institute, 1999.

[90] K. J. Lang, B. A. Pearlmutter, and R. A. Price. Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In *Proceedings ICGI*, volume 1433 of *LNAI*, pages 1–12, 1998.

[91] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *Proceedings ICSE*, pages 591–600. ACM, 2011.

[92] D. Lee and M. Yannakakis. Testing finite-state machines: state identification and verification. *Computers*, 43(3):306–320, 1994.

[93] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings IEEE*, 84(8):1090–1123, 1996.

[94] C. Leita, M. Dacier, F. Massicotte, I. Eurecom, and S. Antipolis. Automatic handling of protocol dependencies and reaction to 0-day attacks with scriptgen based honeypots. In *Proceedings RAID*, pages 185–205, 2006.

[95] C. Leita, K. Mermoud, and M. Dacier. Scriptgen: an automated script generation tool for honeyd. In *Proceedings ACSAC*, pages 203–214. IEEE, 2005.

[96] Z. Li, G. Xia, H. Gao, Y. Tang, Y. Chen, B. Liu, J. Jiang, and Y. Lv. NetShield : Massive semantics-based vulnerability signature matching for high-speed networks state of the art. In *Proceedings SIGCOMM*, pages 279–290. ACM, 2010.

[97] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings NDSS*, 2008.

[98] O. Maler and A. Pnueli. On the learnability of infinitary regular sets. *Information and Computation*, 118(2):316–326, 1995.

[99] C. Meiners, E. Norige, A. X. Liu, and E. Torng. FlowSifter: A counting automata approach to layer 7 field extraction for deep flow inspection. In *Proceedings INFOCOM*, pages 1746–1754. IEEE, 2012.

[100] I.-E. Mens and O. Maler. Learning regular languages over large ordered alphabets. *Logical Methods in Computer Science*, 11, 2015.

[101] Merriam-Webster Dictionary online. System. `https://www.merriam-webster.com/dictionary/system`, date accessed: 2017-8-18.

[102] M. Merten, B. Steffen, F. Howar, and T. Margaria. Next generation LearnLib. In *Proceedings TACAS*, volume 6605 of *LNCS*, pages 220–223. Springer, 2011.

[103] J. Moerman, M. Sammartino, A. Silva, B. Klin, and M. Szynwelski. Learning nominal automata. In *Proceedings POPL*, pages 613–625. ACM, 2017.

[104] E. F. Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[105] L. de Moura and N. Bjørner. Satisfiability modulo theories: Introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[106] J. Narayan, S. K. Shukla, and T. C. Clancy. A survey of automatic protocol reverse engineering tools. *ACM Computing Surveys*, 48(3):40, 2015.

[107] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Molecular Biology*, 48(3):443–453, 1970.

[108] D. Neider. Computing minimal separating DFAs and regular invariants using SAT and SMT solvers. In *Proceedings ATVA*, volume 7561 of *LNCS*, pages 354–369. Springer, 2012.

[109] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: automatic protocol replay by binary analysis. In *Proceedings CCS*, page 311. ACM, 2006.

[110] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings NDSS*, 2005.

[111] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security and Privacy Magazine*, 3(2):58–62, 2005.

[112] J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis*, pages 49–61, 1992.

[113] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: A yacc for writing application protocol parsers. In *Proceedings IMC*, pages 289–300. ACM, 2006.

[114] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In *Proceedings FORTE*, volume 28 of *IFIPAICT*, pages 225–240. Springer, 1999.

[115] A. Petrenko, F. Avellaneda, R. Groz, and C. Oriat. From passive to active FSM inference via checking sequence construction. In *Proceedings ICTSS*, volume 10533 of *LNCS*, pages 126–141. Springer, 2017.

[116] L. Pitt and M. K. Warmuth. The minimum consistent DFA problem cannot be approximated within any polynomial. In *Proceedings STOC*, pages 421–432. ACM, 1989.

[117] H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *Software Tools for Technology Transfer*, 11(5):393–407, 2009.

[118] A. Raman, P. Andreae, and J. Patrick. A beam search algorithm for PFSA inference. *Pattern Analysis and Applications*, 1(2):121–129, 1998.

[119] R. Rivest and R. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[120] R. Rivest and R. Schapire. Diversity-based inference of finite automata. *Journal of the ACM*, 41(3):555–589, 1994.

[121] Z. Rosenberg. Senior F-35 official warns on software breakdowns, relationship crisis, 2012. `https://www.flightglobal.com/news/articles/senior-f-35-official-warns-on-software-breakdowns-relationship-crisis-376590/`, date accessed: 2017-8-18.

[122] J. de Ruiter and E. Poll. Protocol state fuzzing of tls implementations. In *Proceedings USENIX Security*, pages 193–206, 2015.

[123] M. Schuts, J. Hooman, and F. Vaandrager. Refactoring of legacy software using model learning and equivalence checking: An industrial experience report. In *Proceedings IFM*, volume 9681 of *LNCS*, pages 311–325. Springer, 2016.

[124] M. Shahbaz and R. Groz. Inferring Mealy machines. In *Proceedings FM*, volume 5850 of *LNCS*, pages 207–222. Springer, 2009.

[125] G. Shu and D. Lee. Testing security properties of protocol implementations – a machine learning based approach. In *Proceedings ICDCS*, pages 25–25. IEEE, 2007.

[126] W. Smeenk. Applying automata learning to complex industrial software. Master's thesis, Radboud University Nijmegen, September 2012.

[127] W. Smeenk, J. Moerman, F. Vaandrager, and D. N. Jansen. Applying automata learning to embedded control software. In *Proceedings ICFEM*, volume 9407 of *LNCS*, pages 67–83. Springer, 2015.

[128] R. Smetsers, J. Moerman, and D. Jansen. Minimal separating sequences for all pairs of states. In *Proceedings LATA*, volume 9618 of *LNCS*, pages 181–193. Springer, 2016.

[129] I. Sommerville. *Software engineering.* Pearson, 10th edition, 2016.

[130] SPAMfigter News. New Mega-D botnet supersedes Storm, 2008. `http://www.spamfighter.com/News-9799-New-Mega-D-botnet-supersedes-Storm.htm`, date accessed: 2017-8-29.

[131] B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In *Proceedings SFM*, volume 6659 of *LNCS*, pages 256–296. Springer, 2011.

[132] The Linux Foundation. Linux kernel development: how fast it is going, who is doing it, what they are doing, and who is sponsoring it, 2016. `https://www.linuxfoundation.org/publications/linux-kernel-development-2016/`, date accessed: 2017-8-18.

[133] The Linux Kernel Organization. Kernel.org bugzilla. `https://bugzilla.kernel.org`, date accessed: 2017-8-18.

[134] F. Thollard, P. Dupont, and C. de la Higuera. Probabilistic DFA inference using Kullback-Leibler divergence and minimality. In *Proceedings ICML*, pages 975–982, 2000.

[135] C. Thrane, U. Fahrenberg, and K. Larsen. Quantitative analysis of weighted transition systems. *Logic and Algebraic Programming*, 79(7):689–703, 2010.

[136] N. Tishby, F. C. Pereira, and W. Bialek. The information bottleneck method. In *Proceedings Allerton*, pages 368–377, 1999.

[137] B. Trakhtenbrot and Y. M. Barzdin. *Finite automata: behaviour and synthesis.* Elsevier, 1973.

[138] A. Tridgell. How Samba was written, 2003. `https://www.samba.org/ftp/tridge/misc/french_cafe.txt`, date accessed: May 19, 2015.

[139] V. Ulyantsev and F. Tsarev. Extended finite-state machine induction using sat-solver. In *Proceedings ICMLA*, pages 346–349. IEEE, 2011.

[140] F. Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, 2017.

[141] A. Valmari and P. Lehtinen. Efficient minimization of DFAs with partial transition functions. In *Proceedings STACS*, pages 645–656, 2008.

[142] M. P. Vasilevskii. Failure diagnosis of automata. *Cybernetics*, 9(4):653–665, 1973.

[143] S. Verwer. *Efficient identification of timed automata: theory and practice.* PhD thesis, Delft University of Technology, 2010.

[144] N. Walkinshaw, J. Derrick, and Q. Guo. Iterative refinement of reverse-engineered models by model-based testing. In *Proceedings FM*, volume 5850 of *LNCS*, pages 305–320. Springer, 2009.

[145] N. Walkinshaw, B. Lambeau, C. Damas, K. Bogdanov, and P. Dupont. STAMINA: a competition to encourage the development and assessment of software model inference techniques. *Empirical Software Engineering*, 18(4):791–824, 2013.

[146] N. Walkinshaw, R. Taylor, and J. Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, 2016.

[147] Y. Wang, X. Yun, M. Z. Shafiq, L. Wang, A. X. Liu, Z. Zhang, D. Yao, Y. Zhang, and L. Guo. A semantics aware approach to automated reverse engineering unknown protocols. In *Proceedings ICNP*. IEEE, 2012.

[148] Y. Wang, Z. Zhang, D. D. Yao, B. Qu, and L. Guo. Inferring protocol state machine from network traces: A probabilistic approach. In *Proceedings ACNS*, volume 6715 of *LNCS*. Springer, 2011.

[149] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. Reformat : Automatic reverse engineering of encrypted messages. In *Proceedings ESORICS*, pages 200–215, 2009.

[150] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Proceedings NDSS*, pages 1–14, 2008.

[151] J. Yang and D. Evans. Dynamically inferring temporal properties. In *Proceedings PASTE*, pages 23–28. ACM, 2004.

[152] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *Proceedings ICSE*, pages 282–291. ACM, 2006.

[153] M. Zalewski. Binary fuzzing strategies: what works, what doesn't, 2014. `https://lcamtuf.blogspot.nl/2014/08/binary-fuzzing-strategies-what-works.html`, date accessed: 2015-09-15.

[154] M. Zalewski. Fuzzing random programs without execve(), 2014. `https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html`, date accessed: 2015-09-15.

[155] M. Zalewski. American fuzzy lop (AFL) fuzzer, 2015. `http://lcamtuf.coredump.cx/afl/`, date accessed: 2015-09-15.

Curriculum Vitae

2013 – 2017	PhD Computer Science Radboud University
2012 – 2013	MA Human Aspects of Information Sciences Tilburg University, *cum laude*
2008 – 2012	BA Communication & Information Sciences Tilburg University

SIKS Dissertation Series

2011	01	Botond Cseke (RUN), Variational Algorithms for Bayesian Inference in Latent Gaussian Models
	02	Nick Tinnemeier (UU), Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language
	03	Jan Martijn van der Werf (TUE), Compositional Design and Verification of Component-Based Information Systems
	04	Hado van Hasselt (UU), Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference
	05	Bas van der Raadt (VU), Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.
	06	Yiwen Wang (TUE), Semantically-Enhanced Recommendations in Cultural Heritage
	07	Yujia Cao (UT), Multimodal Information Presentation for High Load Human Computer Interaction
	08	Nieske Vergunst (UU), BDI-based Generation of Robust Task-Oriented Dialogues
	09	Tim de Jong (OU), Contextualised Mobile Media for Learning
	10	Bart Bogaert (UvT), Cloud Content Contention
	11	Dhaval Vyas (UT), Designing for Awareness: An Experience-focused HCI Perspective
	12	Carmen Bratosin (TUE), Grid Architecture for Distributed Process Mining
	13	Xiaoyu Mao (UvT), Airport under Control. Multiagent Scheduling for Airport Ground Handling
	14	Milan Lovric (EUR), Behavioral Finance and Agent-Based Artificial Markets

15 Marijn Koolen (UvA), The Meaning of Structure: the Value of Link Evidence for Information Retrieval
16 Maarten Schadd (UM), Selective Search in Games of Different Complexity
17 Jiyin He (UVA), Exploring Topic Structure: Coherence, Diversity and Relatedness
18 Mark Ponsen (UM), Strategic Decision-Making in complex games
19 Ellen Rusman (OU), The Mind's Eye on Personal Profiles
20 Qing Gu (VU), Guiding service-oriented software engineering - A view-based approach
21 Linda Terlouw (TUD), Modularization and Specification of Service-Oriented Systems
22 Junte Zhang (UVA), System Evaluation of Archival Description and Access
23 Wouter Weerkamp (UVA), Finding People and their Utterances in Social Media
24 Herwin van Welbergen (UT), Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior
25 Syed Waqar ul Qounain Jaffry (VU), Analysis and Validation of Models for Trust Dynamics
26 Matthijs Aart Pontier (VU), Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots
27 Aniel Bhulai (VU), Dynamic website optimization through autonomous management of design patterns
28 Rianne Kaptein (UVA), Effective Focused Retrieval by Exploiting Query Context and Document Structure
29 Faisal Kamiran (TUE), Discrimination-aware Classification
30 Egon van den Broek (UT), Affective Signal Processing (ASP): Unraveling the mystery of emotions
31 Ludo Waltman (EUR), Computational and Game-Theoretic Approaches for Modeling Bounded Rationality
32 Nees-Jan van Eck (EUR), Methodological Advances in Bibliometric Mapping of Science
33 Tom van der Weide (UU), Arguing to Motivate Decisions
34 Paolo Turrini (UU), Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations

35 Maaike Harbers (UU), Explaining Agent Behavior in Virtual Training
36 Erik van der Spek (UU), Experiments in serious game design: a cognitive approach
37 Adriana Burlutiu (RUN), Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference
38 Nyree Lemmens (UM), Bee-inspired Distributed Optimization
39 Joost Westra (UU), Organizing Adaptation using Agents in Serious Games
40 Viktor Clerc (VU), Architectural Knowledge Management in Global Software Development
41 Luan Ibraimi (UT), Cryptographically Enforced Distributed Data Access Control
42 Michal Sindlar (UU), Explaining Behavior through Mental State Attribution
43 Henk van der Schuur (UU), Process Improvement through Software Operation Knowledge
44 Boris Reuderink (UT), Robust Brain-Computer Interfaces
45 Herman Stehouwer (UvT), Statistical Language Models for Alternative Sequence Selection
46 Beibei Hu (TUD), Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work
47 Azizi Bin Ab Aziz (VU), Exploring Computational Models for Intelligent Support of Persons with Depression
48 Mark Ter Maat (UT), Response Selection and Turn-taking for a Sensitive Artificial Listening Agent
49 Andreea Niculescu (UT), Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality

---

2012 01 Terry Kakeeto (UvT), Relationship Marketing for SMEs in Uganda
02 Muhammad Umair (VU), Adaptivity, emotion, and Rationality in Human and Ambient Agent Models
03 Adam Vanya (VU), Supporting Architecture Evolution by Mining Software Repositories
04 Jurriaan Souer (UU), Development of Content Management System-based Web Applications

05 Marijn Plomp (UU), Maturing Interorganisational Information Systems
06 Wolfgang Reinhardt (OU), Awareness Support for Knowledge Workers in Research Networks
07 Rianne van Lambalgen (VU), When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions
08 Gerben de Vries (UVA), Kernel Methods for Vessel Trajectories
09 Ricardo Neisse (UT), Trust and Privacy Management Support for Context-Aware Service Platforms
10 David Smits (TUE), Towards a Generic Distributed Adaptive Hypermedia Environment
11 J.C.B. Rantham Prabhakara (TUE), Process Mining in the Large: Preprocessing, Discovery, and Diagnostics
12 Kees van der Sluijs (TUE), Model Driven Design and Data Integration in Semantic Web Information Systems
13 Suleman Shahid (UvT), Fun and Face: Exploring non-verbal expressions of emotion during playful interactions
14 Evgeny Knutov (TUE), Generic Adaptation Framework for Unifying Adaptive Web-based Systems
15 Natalie van der Wal (VU), Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes.
16 Fiemke Both (VU), Helping people by understanding them - Ambient Agents supporting task execution and depression treatment
17 Amal Elgammal (UvT), Towards a Comprehensive Framework for Business Process Compliance
18 Eltjo Poort (VU), Improving Solution Architecting Practices
19 Helen Schonenberg (TUE), What's Next? Operational Support for Business Process Execution
20 Ali Bahramisharif (RUN), Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing
21 Roberto Cornacchia (TUD), Querying Sparse Matrices for Information Retrieval
22 Thijs Vis (UvT), Intelligence, politie en veiligheidsdienst: verenigbare grootheden?
23 Christian Muehl (UT), Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction

24 Laurens van der Werff (UT), Evaluation of Noisy Transcripts for Spoken Document Retrieval
25 Silja Eckartz (UT), Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application
26 Emile de Maat (UVA), Making Sense of Legal Text
27 Hayrettin Gurkok (UT), Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games
28 Nancy Pascall (UvT), Engendering Technology Empowering Women
29 Almer Tigelaar (UT), Peer-to-Peer Information Retrieval
30 Alina Pommeranz (TUD), Designing Human-Centered Systems for Reflective Decision Making
31 Emily Bagarukayo (RUN), A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure
32 Wietske Visser (TUD), Qualitative multi-criteria preference representation and reasoning
33 Rory Sie (OUN), Coalitions in Cooperation Networks (COCOON)
34 Pavol Jancura (RUN), Evolutionary analysis in PPI networks and applications
35 Evert Haasdijk (VU), Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics
36 Denis Ssebugwawo (RUN), Analysis and Evaluation of Collaborative Modeling Processes
37 Agnes Nakakawa (RUN), A Collaboration Process for Enterprise Architecture Creation
38 Selmar Smit (VU), Parameter Tuning and Scientific Testing in Evolutionary Algorithms
39 Hassan Fatemi (UT), Risk-aware design of value and coordination networks
40 Agus Gunawan (UvT), Information Access for SMEs in Indonesia
41 Sebastian Kelle (OU), Game Design Patterns for Learning
42 Dominique Verpoorten (OU), Reflection Amplifiers in self-regulated Learning
43 Withdrawn
44 Anna Tordai (VU), On Combining Alignment Techniques

	45	Benedikt Kratz (UvT), A Model and Language for Business-aware Transactions
	46	Simon Carter (UVA), Exploration and Exploitation of Multilingual Data for Statistical Machine Translation
	47	Manos Tsagkias (UVA), Mining Social Media: Tracking Content and Predicting Behavior
	48	Jorn Bakker (TUE), Handling Abrupt Changes in Evolving Time-series Data
	49	Michael Kaisers (UM), Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions
	50	Steven van Kervel (TUD), Ontologogy driven Enterprise Information Systems Engineering
	51	Jeroen de Jong (TUD), Heuristics in Dynamic Sceduling; a practical framework with a case study in elevator dispatching
2013	01	Viorel Milea (EUR), News Analytics for Financial Decision Support
	02	Erietta Liarou (CWI), MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing
	03	Szymon Klarman (VU), Reasoning with Contexts in Description Logics
	04	Chetan Yadati (TUD), Coordinating autonomous planning and scheduling
	05	Dulce Pumareja (UT), Groupware Requirements Evolutions Patterns
	06	Romulo Goncalves (CWI), The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience
	07	Giel van Lankveld (UvT), Quantifying Individual Player Differences
	08	Robbert-Jan Merk (VU), Making enemies: cognitive modeling for opponent agents in fighter pilot simulators
	09	Fabio Gori (RUN), Metagenomic Data Analysis: Computational Methods and Applications
	10	Jeewanie Jayasinghe Arachchige (UvT), A Unified Modeling Framework for Service Design.
	11	Evangelos Pournaras (TUD), Multi-level Reconfigurable Self-organization in Overlay Services
	12	Marian Razavian (VU), Knowledge-driven Migration to Services

13 Mohammad Safiri (UT), Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly
14 Jafar Tanha (UVA), Ensemble Approaches to Semi-Supervised Learning Learning
15 Daniel Hennes (UM), Multiagent Learning - Dynamic Games and Applications
16 Eric Kok (UU), Exploring the practical benefits of argumentation in multi-agent deliberation
17 Koen Kok (VU), The PowerMatcher: Smart Coordination for the Smart Electricity Grid
18 Jeroen Janssens (UvT), Outlier Selection and One-Class Classification
19 Renze Steenhuizen (TUD), Coordinated Multi-Agent Planning and Scheduling
20 Katja Hofmann (UvA), Fast and Reliable Online Learning to Rank for Information Retrieval
21 Sander Wubben (UvT), Text-to-text generation by monolingual machine translation
22 Tom Claassen (RUN), Causal Discovery and Logic
23 Patricio de Alencar Silva (UvT), Value Activity Monitoring
24 Haitham Bou Ammar (UM), Automated Transfer in Reinforcement Learning
25 Agnieszka Anna Latoszek-Berendsen (UM), Intention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System
26 Alireza Zarghami (UT), Architectural Support for Dynamic Homecare Service Provisioning
27 Mohammad Huq (UT), Inference-based Framework Managing Data Provenance
28 Frans van der Sluis (UT), When Complexity becomes Interesting: An Inquiry into the Information eXperience
29 Iwan de Kok (UT), Listening Heads
30 Joyce Nakatumba (TUE), Resource-Aware Business Process Management: Analysis and Support
31 Dinh Khoa Nguyen (UvT), Blueprint Model and Language for Engineering Cloud Applications
32 Kamakshi Rajagopal (OUN), Networking For Learning; The role of Networking in a Lifelong Learner's Professional Development

33 Qi Gao (TUD), User Modeling and Personalization in the Microblogging Sphere
34 Kien Tjin-Kam-Jet (UT), Distributed Deep Web Search
35 Abdallah El Ali (UvA), Minimal Mobile Human Computer Interaction
36 Than Lam Hoang (TUe), Pattern Mining in Data Streams
37 Dirk Börner (OUN), Ambient Learning Displays
38 Eelco den Heijer (VU), Autonomous Evolutionary Art
39 Joop de Jong (TUD), A Method for Enterprise Ontology based Design of Enterprise Information Systems
40 Pim Nijssen (UM), Monte-Carlo Tree Search for Multi-Player Games
41 Jochem Liem (UVA), Supporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning
42 Léon Planken (TUD), Algorithms for Simple Temporal Reasoning
43 Marc Bron (UVA), Exploration and Contextualization through Interaction and Concepts

---

2014 01 Nicola Barile (UU), Studies in Learning Monotone Models from Data
02 Fiona Tuliyano (RUN), Combining System Dynamics with a Domain Modeling Method
03 Sergio Raul Duarte Torres (UT), Information Retrieval for Children: Search Behavior and Solutions
04 Hanna Jochmann-Mannak (UT), Websites for children: search strategies and interface design - Three studies on children's search performance and evaluation
05 Jurriaan van Reijsen (UU), Knowledge Perspectives on Advancing Dynamic Capability
06 Damian Tamburri (VU), Supporting Networked Software Development
07 Arya Adriansyah (TUE), Aligning Observed and Modeled Behavior
08 Samur Araujo (TUD), Data Integration over Distributed and Heterogeneous Data Endpoints
09 Philip Jackson (UvT), Toward Human-Level Artificial Intelligence: Representation and Computation of Meaning in Natural Language
10 Ivan Salvador Razo Zapata (VU), Service Value Networks

11 Janneke van der Zwaan (TUD), An Empathic Virtual Buddy for Social Support
12 Willem van Willigen (VU), Look Ma, No Hands: Aspects of Autonomous Vehicle Control
13 Arlette van Wissen (VU), Agent-Based Support for Behavior Change: Models and Applications in Health and Safety Domains
14 Yangyang Shi (TUD), Language Models With Meta-information
15 Natalya Mogles (VU), Agent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems: Applications in Safety and Healthcare
16 Krystyna Milian (VU), Supporting trial recruitment and design by automatically interpreting eligibility criteria
17 Kathrin Dentler (VU), Computing healthcare quality indicators automatically: Secondary Use of Patient Data and Semantic Interoperability
18 Mattijs Ghijsen (UVA), Methods and Models for the Design and Study of Dynamic Agent Organizations
19 Vinicius Ramos (TUE), Adaptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support
20 Mena Habib (UT), Named Entity Extraction and Disambiguation for Informal Text: The Missing Link
21 Kassidy Clark (TUD), Negotiation and Monitoring in Open Environments
22 Marieke Peeters (UU), Personalized Educational Games - Developing agent-supported scenario-based training
23 Eleftherios Sidirourgos (UvA/CWI), Space Efficient Indexes for the Big Data Era
24 Davide Ceolin (VU), Trusting Semi-structured Web Data
25 Martijn Lappenschaar (RUN), New network models for the analysis of disease interaction
26 Tim Baarslag (TUD), What to Bid and When to Stop
27 Rui Jorge Almeida (EUR), Conditional Density Models Integrating Fuzzy and Probabilistic Representations of Uncertainty
28 Anna Chmielowiec (VU), Decentralized k-Clique Matching
29 Jaap Kabbedijk (UU), Variability in Multi-Tenant Enterprise Software
30 Peter de Cock (UvT), Anticipating Criminal Behaviour
31 Leo van Moergestel (UU), Agent Technology in Agile Multiparallel Manufacturing and Product Support

32 Naser Ayat (UvA), On Entity Resolution in Probabilistic Data

33 Tesfa Tegegne (RUN), Service Discovery in eHealth

34 Christina Manteli (VU), The Effect of Governance in Global Software Development: Analyzing Transactive Memory Systems.

35 Joost van Ooijen (UU), Cognitive Agents in Virtual Worlds: A Middleware Design Approach

36 Joos Buijs (TUE), Flexible Evolutionary Algorithms for Mining Structured Process Models

37 Maral Dadvar (UT), Experts and Machines United Against Cyberbullying

38 Danny Plass-Oude Bos (UT), Making brain-computer interfaces better: improving usability through post-processing.

39 Jasmina Maric (UvT), Web Communities, Immigration, and Social Capital

40 Walter Omona (RUN), A Framework for Knowledge Management Using ICT in Higher Education

41 Frederic Hogenboom (EUR), Automated Detection of Financial Events in News Text

42 Carsten Eijckhof (CWI/TUD), Contextual Multidimensional Relevance Models

43 Kevin Vlaanderen (UU), Supporting Process Improvement using Method Increments

44 Paulien Meesters (UvT), Intelligent Blauw. Met als ondertitel: Intelligence-gestuurde politiezorg in gebiedsgebonden eenheden.

45 Birgit Schmitz (OUN), Mobile Games for Learning: A Pattern-Based Approach

46 Ke Tao (TUD), Social Web Data Analytics: Relevance, Redundancy, Diversity

47 Shangsong Liang (UVA), Fusion and Diversification in Information Retrieval

---

2015 01 Niels Netten (UvA), Machine Learning for Relevance of Information in Crisis Response

02 Faiza Bukhsh (UvT), Smart auditing: Innovative Compliance Checking in Customs Controls

03 Twan van Laarhoven (RUN), Machine learning for network data

04 Howard Spoelstra (OUN), Collaborations in Open Learning Environments

05 Christoph Bösch (UT), Cryptographically Enforced Search Pattern Hiding

06 Farideh Heidari (TUD), Business Process Quality Computation - Computing Non-Functional Requirements to Improve Business Processes
07 Maria-Hendrike Peetz (UvA), Time-Aware Online Reputation Analysis
08 Jie Jiang (TUD), Organizational Compliance: An agent-based model for designing and evaluating organizational interactions
09 Randy Klaassen (UT), HCI Perspectives on Behavior Change Support Systems
10 Henry Hermans (OUN), OpenU: design of an integrated system to support lifelong learning
11 Yongming Luo (TUE), Designing algorithms for big graph datasets: A study of computing bisimulation and joins
12 Julie M. Birkholz (VU), Modi Operandi of Social Network Dynamics: The Effect of Context on Scientific Collaboration Networks
13 Giuseppe Procaccianti (VU), Energy-Efficient Software
14 Bart van Straalen (UT), A cognitive approach to modeling bad news conversations
15 Klaas Andries de Graaf (VU), Ontology-based Software Architecture Documentation
16 Changyun Wei (UT), Cognitive Coordination for Cooperative Multi-Robot Teamwork
17 André van Cleeff (UT), Physical and Digital Security Mechanisms: Properties, Combinations and Trade-offs
18 Holger Pirk (CWI), Waste Not, Want Not! - Managing Relational Data in Asymmetric Memories
19 Bernardo Tabuenca (OUN), Ubiquitous Technology for Lifelong Learners
20 Lois Vanhée (UU), Using Culture and Values to Support Flexible Coordination
21 Sibren Fetter (OUN), Using Peer-Support to Expand and Stabilize Online Learning
22 Zhemin Zhu (UT), Co-occurrence Rate Networks
23 Luit Gazendam (VU), Cataloguer Support in Cultural Heritage
24 Richard Berendsen (UVA), Finding People, Papers, and Posts: Vertical Search Algorithms and Evaluation
25 Steven Woudenberg (UU), Bayesian Tools for Early Disease Detection

26 Alexander Hogenboom (EUR), Sentiment Analysis of Text Guided by Semantics and Structure

27 Sándor Héman (CWI), Updating compressed colomn stores

28 Janet Bagorogoza (TiU), Knowledge Management and High Performance; The Uganda Financial Institutions Model for HPO

29 Hendrik Baier (UM), Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains

30 Kiavash Bahreini (OU), Real-time Multimodal Emotion Recognition in E-Learning

31 Yakup Koç (TUD), On the robustness of Power Grids

32 Jerome Gard (UL), Corporate Venture Management in SMEs

33 Frederik Schadd (TUD), Ontology Mapping with Auxiliary Resources

34 Victor de Graaf (UT), Gesocial Recommender Systems

35 Jungxao Xu (TUD), Affective Body Language of Humanoid Robots: Perception and Effects in Human Robot Interaction

---

2016 01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines

02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow

03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support

04 Laurens Rietveld (VU), Publishing and Consuming Linked Data

05 Evgeny Sherkhonov (UVA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers

06 Michel Wilson (TUD), Robust scheduling in an uncertain environment

07 Jeroen de Man (VU), Measuring and modeling negative emotions for virtual training

08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data

09 Archana Nottamkandath (VU), Trusting Crowdsourced Information on Cultural Artefacts

10 George Karafotias (VUA), Parameter Control for Evolutionary Algorithms

11 Anne Schuth (UVA), Search Engines that Learn from Their Users

12 Max Knobbout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
13 Nana Baah Gyan (VU), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization
15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
16 Guangliang Li (UVA), Socially Intelligent Autonomous Agents that Learn from Human Reward
17 Berend Weel (VU), Towards Embodied Evolution of Robot Organisms
18 Albert Meroño Peñuela (VU), Refining Statistical Data on the Web
19 Julia Efremova (Tu/e), Mining Social Structures from Genealogical Data
20 Daan Odijk (UVA), Context & Semantics in News & Web Search
21 Alejandro Moreno Célleri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground
22 Grace Lewis (VU), Software Architecture Strategies for Cyber-Foraging Systems
23 Fei Cai (UVA), Query Auto Completion in Information Retrieval
24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior
26 Dilhan Thilakarathne (VU), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
27 Wen Li (TUD), Understanding Geo-spatial Information on Social Media
28 Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control
29 Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
30 Ruud Mattheij (UvT), The Eyes Have It
31 Mohammad Khelghati (UT), Deep web content monitoring

	32	Eelco Vriezekolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations
	33	Peter Bloem (UVA), Single Sample Statistics, exercises in learning from just one example
	34	Dennis Schunselaar (TUE), Configurable Process Trees: Elicitation, Analysis, and Enactment
	35	Zhaochun Ren (UVA), Monitoring Social Media: Summarization, Classification and Recommendation
	36	Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
	37	Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry
	38	Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design
	39	Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
	40	Christian Detweiler (TUD), Accounting for Values in Design
	41	Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance
	42	Spyros Martzoukos (UVA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
	43	Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
	44	Thibault Sellam (UVA), Automatic Assistants for Database Exploration
	45	Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
	46	Jorge Gallego Perez (UT), Robots to Make you Happy
	47	Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks
	48	Tanja Buttler (TUD), Collecting Lessons Learned
	49	Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis
	50	Yan Wang (UVT), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains
2017	01	Jan-Jaap Oerlemans (UL), Investigating Cybercrime

02 Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation
03 Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines
04 Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store
05 Mahdieh Shadi (UVA), Collaboration Behavior
06 Damir Vandic (EUR), Intelligent Information Systems for Web Product Search
07 Roel Bertens (UU), Insight in Information: from Abstract to Anomaly
08 Rob Konijn (VU) , Detecting Interesting Differences:Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery
09 Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text
10 Robby van Delden (UT), (Steering) Interactive Play Behavior
11 Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment
12 Sander Leemans (TUE), Robust Process Mining with Guarantees
13 Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology
14 Shoshannah Tekofsky (UvT), You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior
15 Peter Berck (RUN), Memory-Based Text Correction
16 Aleksandr Chuklin (UVA), Understanding and Modeling Users of Modern Search Engines
17 Daniel Dimov (UL), Crowdsourced Online Dispute Resolution
18 Ridho Reinanda (UVA), Entity Associations for Search
19 Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval
20 Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility
21 Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)
22 Sara Magliacane (VU), Logics for causal inference under uncertainty