JMLR: Workshop and Conference Proceedings 34:109–123, 2014 Proceedings of the 12th ICGI

Learning Nondeterministic Mealy Machines

Ali Khalili ALI.KHALILI@EDU.UNIGE.IT
Armando Tacchella ARMANDO.TACCHELLA@UNIGE.IT
Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS)
Università degli Studi di Genova - Via Opera Pia 13 - 16145 Genova, Italy

Editors: Alexander Clark, Makoto Kanazawa and Ryo Yoshinaka

Abstract

In applications where abstract models of reactive systems are to be inferred, one important challenge is that the behavior of such systems can be inherently nondeterministic. To cope with this challenge, we developed an algorithm to infer nondeterministic computation models in the form of Mealy machines. We introduce our approach and provide extensive experimental results to assess its potential in the identification of black-box reactive systems. The experiments involve both artificially-generated abstract Mealy machines, and the identification of a TFTP server model starting from a publicly-available implementation.

1. Introduction

Automata-based inference, i.e., the ability to infer abstract models of black-box software components as automata, is a vibrant research and application area — see, e.g., Walkinshaw et al. (2013) for the results of a recent competition among model-inference engines. One important challenge in this arena is that reactive systems are often inherently nondeterministic. Since it is impossible to track and control all possible sources of external influence on a complex reactive system, one may consider nondeterministic models as a solution to provide concise and usable abstractions. Indeed, while there is a substantial body of work about learning automata, including comprehensive tools such as LearnLib Raffelt et al. (2009), they focus on deterministic automata and transducers, limiting their effective applicability to many reactive systems. Our research aims to fill this gap by enabling the inference of nondeterministic models for black-box reactive systems. The core of our contribution is the algorithm N^*, an extension and systematization of works by Shahbaz and others — see Shahbaz (2008); El-Fakih et al. (2010) — to infer nondeterministic Mealy machines. Using our tool, AIDE[1] , we have conducted an experimental campaign to evaluate N^* considering various features of the target machines. In particular, we implemented a generator and emulator of random Mealy machines, wherein we can control various parameters. Knowing the target machine in advance, we can compare "ideal" and "real" implementation of N^*. In the ideal case, N^* acquires all possible outcomes of a nondeterministic transition with a single query, and it performs precise equivalence checks between conjectures and the target automaton. In the real case, multiple queries and approximate equivalence checks are required instead, causing a decrease in performances that we can assess in a quantitative

1. AIDE (Automata IDentification Engine) is an open-source software written in C#. It includes an implementation of N^*. The code and documentations are available at https://aide.codeplex.com/.

way. As a further assessment of practical feasibility, we have evaluated N^* on a working implementation of a TFTP client/server protocol.

1.1. Background

An *alphabet* Σ is a finite set of symbols, and Σ^* is the transitive, symmetric and reflexive closure of Σ; we call *word* a member of Σ^*, and we denote with $|w|$ the *length* of w, i.e., the number of symbols in w; ϵ is the *empty word*, such that $\epsilon \in \Sigma^*$ for all Σ and $|\epsilon| = 0$; we write Σ^+ to denote $\Sigma^* \setminus \{\epsilon\}$; given $u, v \in \Sigma^*$ and $a \in \Sigma$, we have that $a.v \in \Sigma^*$ and $v.a \in \Sigma^*$ are the words obtained by prefixing and postfixing a to v, respectively; we abuse notation and write $u.v$ to denote the word obtained by appending v to u. A *prefix* of a word $w \in \Sigma^*$ is a word $u \in \Sigma^*$ such that $w = u.v$ for some word $v \in \Sigma^*$. Similarly, a *suffix* of a word $w \in \Sigma^*$ is a word $v \in \Sigma^*$ such that $w = u.v$ for some $u \in \Sigma^*$. We say that a set of words $S \subseteq \Sigma^*$ is *prefix-closed* (resp. *suffix-closed*) exactly when, for every word $w \in S$, S contains all possible prefixes (resp. suffixes) of w. We denote with $\mathit{Suff}(w)$ the suffix-closed set comprised of all the suffixes of w, and $\mathit{Pref}(h)$ the prefix-closed set comprised of all the prefixes of h.

A *Deterministic Finite-state Mealy machine* (that we abbreviate as DFM) is defined as a quintuple $M = (Q, \Sigma_I, \Sigma_O, q_0, \tau)$, where Σ_I is the input alphabet and Σ_O is the output alphabet; Q is a set of *states*, $q_0 \in Q$ is the *initial state*, and $\tau : Q \times \Sigma_I \to Q \times \Sigma_O$ is the transition function. The sets Σ_I, Σ_O, and Q are finite, non-empty and mutually disjoint. We consider two projections of τ, namely the *next state function* $\delta_\tau : Q \times \Sigma_I \to Q$, and the *output function* $\lambda_\tau : Q \times \Sigma_I \to \Sigma_O$. The extension of λ_τ to words is denoted as $\lambda^*_\tau : Q \times \Sigma^*_I \to \Sigma^*_O$, and it is defined recursively for all $q \in Q$ as

$$\lambda^*_\tau(q, w) = \begin{cases} \epsilon & \text{if } w = \epsilon \\ \lambda_\tau(q, a).\lambda^*_\tau(\delta_\tau(q, a), v) & \text{if } w = a.v,\ a \in \Sigma_I,\ v \in \Sigma^*_I. \end{cases}$$

The (*input-output*) *relation* $R_M : \Sigma^*_I \to \Sigma^*_O$ computed by M can be defined as the set of all words $v \in \Sigma^*_O$ such that $v = \lambda^*(q_0, u)$ for some $u \in \Sigma^*_I$. Notice that for every $u \in \Sigma^*_I$ and corresponding $v = R_M(u)$ we have that $|u| = |v|$, i.e., DFMs define one-by-one translations.

A *Nondeterministic Finite-state Mealy machine* (called NFM here) is a quintuple $M = (Q, \Sigma_I, \Sigma_O, q_0, \tau)$, where Σ_I, Σ_O, Q and q_0 are defined as in DFMs, whereas the transition relation is now defined as $\tau : Q \times \Sigma_I \to 2^{Q \times \Sigma_O}$. Intuitively, in NFMs, the same state and input symbol may result in one or more different *pairs* of state and symbol. The projections of τ on states and outputs are $\delta_\tau : Q \times \Sigma_I \to 2^Q$ and $\lambda_\tau : Q \times \Sigma_I \to 2^{\Sigma_O}$, respectively. The extension of δ_τ to words, denoted as $\delta^*_\tau : Q \times \Sigma^*_I \to 2^Q$, can be defined recursively for all $q \in Q$ as

$$\delta^*_\tau(q, w) = \begin{cases} q & \text{if } w = \epsilon \\ \bigcup_{q' \in \delta_\tau(q,a)} \delta^*_\tau(q', v) & \text{if } w = a.v,\ a \in \Sigma_I,\ v \in \Sigma^*_I. \end{cases}$$

The extension of λ_τ to words is denoted as $\lambda^*_\tau : Q \times \Sigma^*_I \to 2^{\Sigma^*_O}$, and it can be defined as follows. Given $w \in \Sigma^*_I$ and $v \in \Sigma^*_O$, we have that $v \in \lambda^*_\tau(q, w)$ for some state $q \in Q$ if and only if either $w = v = \epsilon$, or $w = a.w'$, $v = x.v'$ for some $a \in \Sigma_I$ and $x \in \Sigma_O$, and there exists a state q' such that $(q', x) \in \tau(q, a)$ and $v' \in \lambda^*_\tau(q', w')$. Intuitively, given an input word w and a state q, the function $\delta^*_\tau(q, w)$ defines the "frontier" of states reachable from q given the input word w, whereas $\lambda^*_\tau(q, w)$ defines the set of output words "seen" on each

path from q to any of the states in $\delta^*_\tau(q, w)$. In the case of a NFM M, the input-output relation $R_M : \Sigma_I^* \to 2^{\Sigma_O^*}$ is defined as the set of all words $v \in \Sigma_O^*$ such that $v = \lambda^*_\tau(q_0, u)$ for some $u \in \Sigma_I^*$. Two NFMs $M_1 = (Q_1, \Sigma_I, \Sigma_O, q_{01}, \tau_1)$ and $M_2 = (Q_2, \Sigma_I, \Sigma_O, q_{02}, \tau_2)$ are said to be *behaviorally equivalent* iff for all $u \in \Sigma_I^*$, we have that $\lambda^*_{\tau_1}(q_{01}, u) \equiv \lambda^*_{\tau_2}(q_{02}, u)$.

1.2. Related Works

Our work is relevant in the context of active learning in the sense pioneered by Gold in Gold (1972), and later refined by Angluin in Angluin (1987) with her L^* algorithm. Further related contributions along the line of Angluin's L^* include Bollig et al. (2009), wherein the algorithm NL^* is introduced. NL^* can learn a subclass of non-deterministic finite state automata (NFAs) called *residual finite-state automata*. The advantage of using NL^* is that, for regular languages, NFAs are usually more compact than equivalent DFAs, which could make NL^* a preferable choice in many learning applications. Even if NL^* computes non-deterministic machines as conjectures, the target system is assumed to be deterministic. Therefore, the implementation of membership and equivalence queries is similar to L^*, and it does not deal with intrinsic target non-determinism as we do.

An adaptation of Angluin's L^* algorithm for identifying DFMs was first developed in Niese (2003) and it was further extended by Shahbaz in Shahbaz (2008), and by Irfan Irfan (2012). These contributions are all based on Angluin's concept of *observation table*, i.e., a data structure which is processed by the identification algorithm to collect the observations, and to formulate the conjecture about the target system. Alternative approaches to implicit identification through observation tables include, e.g., Merten et al. (2012) where the authors describe an identification algorithm for DFMs that directly constructs a state machine hypothesis according to observations. Observation pack Howar (2012) is another algorithm that can be instantiated with different strategies for handling counterexamples.

None of the algorithms cited above can deal with the problem of inferring NFMs and, to the best of our knowledge, the only proposal extending L^*-type learning to non-deterministic machines is sketched in El-Fakih et al. (2010). In spirit, the extension therewith proposed is very similar to the one we discuss in Section 2. However, since El-Fakih et al. (2010) is just an extended abstract, it does not provide either detailed technical descriptions or the kind of extensive empirical results that we present in the remainder of this paper. Furthermore, no detailed follow-up is to be found in the literature with the exception of our own contribution.

2. Learning NFMs

NFMs can encode a larger set of input-output relations than those expressible with DFMs, i.e., given an NFM M, it may be impossible to obtain a *behaviorally equivalent* DFM M' such that, for all $u \in \Sigma_I^*$, we have that $R_{M'}(u) \in R_M(u)$ and, conversely, every $v \in R_M(u)$ is such that $R_{M'}(u) = v$. A counterexample of this general statement is easily obtained considering a NFM $M = (Q, \Sigma_I, \Sigma_O, q_0, \tau)$ such that the input alphabet is $\Sigma_I = \{a\}$, the output alphabet is $\Sigma_O = \{x, y\}$, and the set of states is $Q = \{q_0, q_1\}$; and $\tau(q_0, a) = \{(q_1, x), (q_1, y)\}$. The relation R_M is thus fully specified by $R_M(a) = \{x, y\}$. If we assume that there exists a DFM $M' = (Q', \Sigma_I, \Sigma_O, q'_0, \tau')$ such that $R_{M'}$ coincides with R_M, then for every input word $w \in \Sigma_I^*$, we must have that $\lambda^*_\tau(q_0, w) = \lambda^*_{\tau'}(q'_o, w)$. The machine M' must have one initial state q'_0, in which a transition on input a must be enabled. Since δ_τ is already a

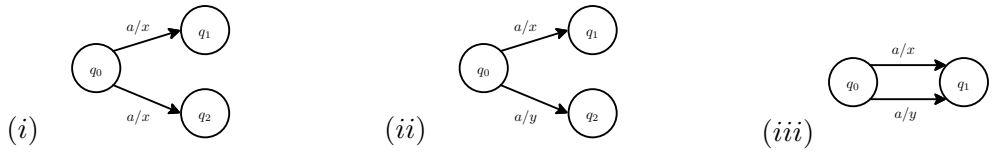


Figure 1: Nondeterminism in a NFM; in the patterns above, we assume that $M = (Q, \Sigma_I, \Sigma_O, q_0, \tau)$ with $a \in \Sigma_I$, $x, y \in \Sigma_O$ where $x \neq y$, and $q_0, q_1, q_2 \in Q$. An edge labeled by a/x from q_i to q_j means that $q_j \in \delta_\tau(q_i, a)$ and $x \in \lambda_\tau(q_i, a)$.

function, $\delta_{\tau'}$ is a "copy" of δ_τ, i.e., $\delta_{\tau'}(q'_0, a) = q'$ where $q' \in Q'$ is a "fresh" state in Q', i.e., $q' \neq q'_0$, and there are no other states in Q'. Since there is only state q'_0 where the symbol a is accepted as input, we should have $\lambda_{\tau'}(q'_0, a) = x$ and $\lambda_{\tau'}(q'_0, a) = y$, which contradicts the assumption that M' is deterministic — see, e.g., Figure 1 (iii).

Depending on the definition of τ, NFMs can show different patterns of nondeterminism. As shown pictorially in Figure 1, we can consider three basic patterns. In pattern (i) we have that δ_τ is not a function whereas λ_τ is a function; in pattern (ii), both δ_τ and λ_τ are not functions; finally, in pattern (iii) we have that only λ_τ is not a function. We say that NFMs exhibit *state nondeterminism* when δ_τ is not a function, and *output nondeterminism* when λ_τ is not a function. We call an NFM M as *identifiable* when, given each input word $w \in \Sigma_I^+$ and a corresponding output word $u \in \lambda_\tau^*(q_0, w)$, there exists a unique sequence of states $q_0, q_1, \ldots q_n$ such that $q_i = \delta_\tau(q_{i-1}, w_i)$ and $\lambda_\tau(q_{i-1}, w_i) = u_i$ for all $1 \leq i \leq n$, where w_i denotes the i-th symbol in the word w. If a machine shows only output nondeterminism, then it is also identifiable by definition, whereas state nondeterminism does not necessarily preserve identifiability. For identifiable machines, we can introduce a concept of minimality, which lays the foundation for our identification algorithm. We say that an identifiable NFM $M = (Q, \Sigma_I, \Sigma_O, q_0, \tau)$ is *minimal* when there does not exits another NFM $M' = (Q', \Sigma_I, \Sigma_O, q'_0, \tau')$ such that M and M' are behaviorally equivalent but $|Q_{M'}| < |Q_M|$.

Given a target black-box implementation, in order to learn its NFM model $M = (Q, \Sigma_I, \Sigma_O, q_0, \tau)$, we assume that the input alphabet Σ_I is known in advance, the target implementation can be reset before executing each query, and either M is identifiable,

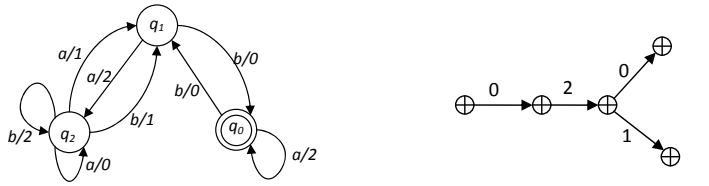
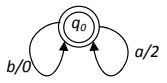


Figure 2: On the left, an NFM with three states, input alphabet {a,b}, and output alphabet {0,1,2}; q_0 is the initial state and the relation τ is represented by directed edges labeled with the corresponding input/output symbols. On the right, an example of an output tree.

	a	b
$\langle\rangle$	2	0
$\langle a,2\rangle$	2	0
$\langle b,0\rangle$	2	0



	a	b	$a.a$
$\langle\rangle$	2	0	2.2
$\langle a,2\rangle$	2	0	2.2
$\langle b,0\rangle$	2	0	2.1,2.2

Figure 3: The initial observation table (left); corresponding one state conjecture NFM (middle); table after adding the first counterexample (right).

or there exists a (behaviorally equivalent) identifiable machine[2]. We further assume that queries posed by the learning algorithm are answered by a *Minimum Adequate Teacher* (MAT) Angluin (1987) (also called *oracle*) which knows perfectly the structure of M. In order to exemplify the various phases of the algorithm, we consider the simple NFM depicted in Figure 2 (left). For simplicity, the NFM has only one nondeterministic transition, namely from state "q_2" on the input symbol "a", but such transition is enough to prevent any DFM learner to accomplish its task.

We consider elements of the set $\Sigma_{IO}^* = (\Sigma_I \times \Sigma_O)^*$ to represent potential states in Q. To avoid ambiguity with input and output words, we call *sequences* the members of Σ_{IO}^*, and we let $\langle\,\rangle$ denote the *empty sequence*; given $h \in \Sigma_{IO}^*$ with $h = \langle(a_1, x_1), \ldots, (a_n, x_n)\rangle$, we define its *input projection* $h_I \in \Sigma_I^*$ as $h_I = a_1 \ldots a_n$, and similarly its *output projection* $h_O \in \Sigma_O^*$ as $h_O = x_1 \ldots x_n$. Other non-specific definitions and notations are used interchangeably between words and sequences. We say that $s \in \Sigma_{IO}^*$ is an *access sequence* when it uniquely identifies a potential state in M. For example, the sequence $\langle(b,0),(a,2)\rangle$ is an access sequence for the state "q_2" in the NFM of Figure 2. As for transitions, given a starting state q and a word $w \in \Sigma_I^*$, we could have that $|\lambda_\tau^*(q,w)| > 1$, i.e., there are many possible output words corresponding to w when starting from q in M. For this reason, we consider prefix trees of symbols in Σ_O, where each edge is an output symbol, and each path from the root to a leaf is a word in Σ_O^*. Prefix trees yield a compact representation of the set of output words $\lambda_\tau^*(q,w)$ because all words having some prefix in common share some subpath from the root to the leaf of the tree. For example, in the NFM of Figure 2, if $q = q_0$ and $w = baa$, then $\lambda_\tau^*(q,w) = \{021, 020\}$ corresponding to the tree in Figure 2 (right).

Our N^* algorithm organizes accumulated information about states and transitions in an *observation table*. Formally, an observation table is denoted by $\Gamma_n = (S_n, E_n, T_n)$ where $S_n \subseteq \Sigma_{IO}^*$ is a prefix-closed set of access sequences; $E_n \subseteq \Sigma_I^+$ is a suffix-closed set of input words; T_n is a function $(S_n \cup S_n') \times E_n \to 2^{\Sigma_O^+}$, where $S_n' \subseteq S_n.\Sigma_{IO}$; given the sequence $s \in (S_n \cup S_n')$, the input word $e \in E_n$, and assuming that q is the state reached after observing the sequence s, then $T_n(s,e)$ contains $\lambda_\tau^*(q,e)$ represented as a prefix tree whose edges are labeled with symbols from Σ_O. Two examples of observation table contents generated by N^* when learning the NFM of Figure 2, knowing the input alphabet $\Sigma_I = \{a, b\}$, are shown in Figure 3. In the following, we will detail their contents across various phases of N^*.

The observation table is the basis to extract a conjecture about the unknown NFM M but in order to extract a valid conjecture, its contents must satisfy specific properties. Let

2. It could be proven that for any NFM, there is a minimal behavior-equivalent identifiable NFM. But this is out of the scope of the current paper.

$s, t \in S_n \cup S'_n$ be two rows in Γ_n; we say that s and t are *equivalent*, denoted by $s \cong_{E_n} t$, if and only if $T_n(s, e) = T_n(t, e)$ for all $e \in E_n$. To extract a valid conjecture C_n from Γ_n, two conditions must hold. The first one is that Γ_n must be *closed*, i.e., for each $t \in S'_n$, there exists an $s \in S_n$ such that $s \cong_{E_n} t$. If Γ_n is not closed, then there is a state $s \in S_n$ and $(a, x) \in \Sigma_{IO}$ such that a transition cannot be defined for s, input a, and corresponding output x. The second one is that Γ_n must be *consistent*, i.e., for each $s, t \in S_n$ such that $s \neq t$, if $s \cong_{E_n} t$ holds, then for all $a \in \Sigma_I$ and $x \in T(s, a)$ we have also $s.(a, x) \cong_{E_n} t.(a, x)$. If Γ_n is not consistent, then two seemingly equivalent states may yield different states for the same $(a, x) \in \Sigma_{IO}$. A conjecture $C_n = (Q_n, \Sigma_I, \Sigma_O, q_{n0}, \tau_n)$ is extracted from a closed and consistent Γ_n by defining $Q_n = S_n$. Given $s \in S_n \cup S'_n$, let $[s] = \{t \in S_n \mid t \cong_{E_n} s\}$ denote the *equivalence class* of s. Then, the initial state is the equivalence class of the empty sequence, i.e., $q_{n0} = [\langle\,\rangle]$, and the transition relation is defined as

$$\tau_n([s], a) = \{([s.(a, x)], x) \mid x \in T_n(s, a)\}$$

for all $s \in S_n$ and $a \in \Sigma_I$. Notice that, unless all output transitions have been explored, the conjecture C_n cannot be equivalent to the unknown machine.

The complete pseudo-code of algorithm N^* is presented in Figure 4. Given the input alphabet Σ_I, and a MAT Ω, the observation table Γ_n is initialized (lines 4-13), and then filled by asking suitable queries to Ω (lines 16-25). Whenever Γ_n is closed, a conjecture C_n is extracted from Γ_n, and Ω is queried about equivalence of such conjecture to the unknown NFM (lines 26-27). When the equivalence test fails, the counterexample (cex) triggers updates of the observation table (lines 28-34). The main loop (lines 15-38) is repeated until the equivalence query succeeds. As we can see in Figure 4, there are two subroutines to interface N^* with Ω, namely outputQuery and equivalenceQuery. outputQuery implements the function $\rho_\Omega : \Sigma^*_{IO} \times \Sigma^+_I \to 2^{\Sigma^+_O}$ defined as detailed in the following. Let $\mu_\tau : Q \times \Sigma_{IO} \to 2^Q$ be a function such that $\mu_\tau(q, (a, x)) = \{q_1, \ldots, q_n\}$ exactly when $\tau(q, a) = \{(q_1, x), \ldots, (q_2, x)\}$, with $a \in \Sigma_I$, $x \in \Sigma_O$, $q, q_1, \ldots, q_n \in Q$. Intuitively, $\mu_\tau(q, (a, x))$ returns

```
1: Input: input alphabet Σ_I, MAT Ω
2: Output: NFM conjecture C_n
3:
4: Initialize S_n ← {⟨⟩}, S'_n ← ∅, E_n ← Σ_I
5: Initialize Γ_n ← (S_n, E_n, ⊥).
6: for all a ∈ Σ_I do
7:     O_a ← outputQuery(Ω, ⟨⟩, i); Add O_a to T_n(⟨⟩, a)
8:     for all x ∈ O_a do
9:         S'_n ← S'_n ∪ {(a, x)}
10:        for all b ∈ Σ_I do
11:            Add outputQuery(Ω, (a, x), b) to T_n((a, x), b)
12:        end for
13:    end for
14: end for
15: while (true) do
16:     while (Γ_n is not closed) do
17:         find t ∈ S'_n s.t. there is no s ∈ S_n s.t. t ≅_{E_n} s
18:         S_n ← S_n ∪ {t}; S'_n ← S'_n \ {t}
19:         for all a ∈ Σ_I, x ∈ T_n(t, a) do
20:             S'_n ← S'_n ∪ {t.(a, x)}
21:             for all w ∈ E_n do
22:                 Add outputQuery(Ω, t.(a, x), w) to T_n(t.(a, x), w)
23:             end for
24:         end for
25:     end while
26:     Extract conjecture C_n from Γ_n
27:     if equivalenceQuery(Ω, C_n) returns cex h ∈ Σ^+_IO then
28:         h ← u.v where u is the longest seq. s.t. u ∈ (S_n ∪ S'_n).
29:         for all w ∈ Suff(v_I) do
30:             E_n ← E_n ∪ {w}
31:             for all t ∈ S_n ∪ S'_n do
32:                 Add outputQuery(Ω, t, w) to T_n(t, w)
33:             end for
34:         end for
35:     else
36:         return C_n
37:     end if
38: end while
```

Figure 4: Algorithm N^* for learning NFMs.

all the states that are reachable from q along a single transition labeled a/x, i.e., all the states reached when reading input a and emitting output x. The extension $\mu^*_\tau : Q \times \Sigma^*_{IO} \to 2^Q$ to sequences, for $s \in \Sigma^*_{IO}$, is defined as

$$\mu^*_\tau(q, s) = \begin{cases} q & \text{if } s = \langle\,\rangle \\ \bigcup_{q' \in \mu_\tau(q,(x,a))} \mu^*_\tau(q', t) & \text{if } s = (x, a).t,\ (a, x) \in \Sigma_{IO} \end{cases} \tag{1}$$

The result of μ^*_τ is the set of states which are reachable from q, assuming that the symbols in s_I are given as input, and the symbols in s_O are seen as output. Notice that, if M is identifiable, then $|\mu_\tau(q, (a, x))| \leq 1$ and $|\mu^*_\tau(q, s)| \leq 1$, for every $q \in Q$, $a \in \Sigma_I$, $x \in \Sigma_O$ and $s \in \Sigma^*_{IO}$. Therefore, for an identifiable machine, if s is an access sequence to some state, i.e., $\mu^*_\tau(q_0, s) \neq \emptyset$, then we can write $\rho_\Omega(s, w) = \lambda^*_{\tau_\Omega}(\mu^*_{\tau_\Omega}(q_0, s), w)$, where $s \in \Sigma^*_{IO}$, $w \in \Sigma^*_I$, and τ_Ω is the transition relation of the unknown NFM supplied by the oracle Ω. If s is not an access sequence, i.e., $\mu^*_\tau(q_0, s) = \emptyset$, then it means that for some pair (a, x) in s, either a is not accepted, or the observed output x' is such that $x' \neq x$. In these cases, we assume that $\rho_\Omega(s, w) = \perp$. Informally speaking, given an access sequence s, outputQuery will bring the unknown machine M from the initial state q_0 to the state $\mu^*_{\tau_\Omega}(q_0, s)$, i.e., the state uniquely identified by s; then, starting from this state, outputQuery will return all possible output words emitted by M on the input w. Here, it is important to highlight that all the results above are based on the identifiability assumption and the use of access sequences rather than access words. equivalenceQuery checks whether the conjecture C_n corresponds to the unknown machine M. If $R_M : \Sigma^*_I \to 2^{\Sigma^*_O}$ is the unknown input-output relation, and $R_{C_n} : \Sigma^*_I \to 2^{\Sigma^*_O}$ is the relation based on the conjecture supplied to equivalenceQuery, then the result is either ϵ – meaning that M and C_n are behaviorally equal – or some other non-empty sequence $h \in \Sigma^+_{IO}$ such that $R_M(h_I) \not\equiv R_{C_n}(h_I)$. Notice that $R_M(h_I)$ may contain yet-to-be-seen output symbols, but this is not an issue because N^* can acquire them through the results of outputQuery performed in line 32 of N^*.

A fundamental property of N^* is that Γ_n is always consistent, and thus the only condition that must be checked before testing a conjecture C_n is that Γ_n is closed. Based on this fact, N^* can be shown to always terminate and identify unknown NFMs correctly. As for the complexity of N^*, we provide an upper bound on the space used by N^* to store Γ_n. Given an unknown –minimal– NFM $M = (Q, \Sigma_I, \Sigma_O, q_0, \tau)$, the bound is defined in terms of the following parameters

$$\chi = \sum_{q \in Q} \sum_{a \in \Sigma_I} |\tau(q, a)| \qquad \beta = \max_{q \in Q, a \in \Sigma_I} \{|\tau(q, a)|\}$$

where χ represents the total number of transitions in M, and β represents the maximum cardinality of τ in a given state, for a given input symbol. Intuitively, β represents the "maximum amount of nondeterminism" that can be found in M. If l is the maximum length of any counterexample provided by equivalenceQuery, the number of cells in Γ_n is $|S_n \cup S'_n| \cdot |E_n|$. Considering a column $e \in E_n$ in the observation table Γ_n, we notice that each cell stores the result of some output query as a tree of depth $|e|$ where in the maximum number of nodes is $(\beta^{|e|+1} - 1)/(\beta - 1) = O(\beta^{|e|})$. Therefore, the upper bound on the size of Γ_n is:

$$(\chi + 1) \cdot (|\Sigma_I| \cdot \beta + (|Q| - 1) \cdot \beta^{l+1}) \tag{2}$$

Notice that, in case of DFMs, output trees would have just one leaf ($\beta = 1$), and thus the bound becomes polynomial in the size of M, which corresponds to the findings of Shahbaz (2008). On the other hand, the complexity bound of NFMs will depend exponentially on the amount of nondeterminism exhibited by the target machine.

A working implementation of N^* — as well as any other MAT-based learning algorithm — requires us to consider testing techniques in order to approximate output and equivalence queries. Given an unknown NFM $M = (Q, \Sigma_I, \Sigma_O, q_o\tau)$, a test sequence $u \in \Sigma_I^*$ results in some string $v \in O_u$ where $O_u \subseteq \Sigma_O^*$. To build a correct conjecture, N^* must know O_u in its entirety and, in practice, this is possible only if we assume that repeatedly supplying the string u to M is sufficient to observe all the elements of O_u, eventually. We call this *no rare events* (NRE) hypothesis, since it amounts to assume that all nondeterministic behaviors can be elicited in a finite amount of time, if repeated tests are executed. Notice that we do not model the probability of different transitions on the same input symbol, but rather we consider them as nondeterministic ones. Indeed, N^* assumes that the output trees returned by outputQuery are exact, i.e., they include all possible outputs resulting from a given input stimulus. For this reason, an incomplete result from outputQuery, may cause N^* to fail, as we discuss further in Section 3. In this sense, our approach is different from probably approximate correct (PAC) learning Valiant (1984) because we are not interested in approximating the target automaton, but rather in learning a behaviorally-equivalent (minimal) model of it. Let us assume that from some state $q \in Q$ there exist a number of nondeterministic transitions on a symbol $a \in \Sigma_I$, i.e., $\tau(q, a) = \{(q_1, x_1), \ldots, (q_n, x_n)\}$, where $q_i \in Q$ and $x_i \in \Sigma_O$ for all $1 \leq i \leq n$. In its tightest interpretation, the NRE hypothesis ensures that the probability of observing some symbol x_i, given the state q and input a, is $\frac{1}{n}$ for all $1 \leq i \leq n$. Therefore, after m trials, the probability of *not* observing a specific output symbol from that state is $p(n, m) = [1 - (1/n)]^m$. With this interpretation, the practical consequence of the NRE hypothesis is that $p(n, m)$ decays sharply with m, if the values of n are relatively small. If we lift this uniform probability requirement, we can still consider the smallest probability $p > 0$ of observing some symbol, and thus testing m times for an input yields a probability $(1 - p)^m$ of *not* observing that symbol. Given an additional confidence parameter c, i.e., the probability of observing all the transitions, then $r(c, p) = \log(1 - c)/\log(1 - p)$ repetitions are required to match the confidence level. The confidence c and the probability p are thus the parameters of outputQuery from which we can set the value of repetitions $r(c, p)$.

As for equivalence queries, there is an extensive literature on different model traversal algorithms to produce conformance test sequences for deterministic systems – see, e.g., Lee and Yannakakis (1996). However, if the black-box system is nondeterministic, given a state and an input, more than one move might be possible, and thus the test case is not just a sequence of inputs, but rather a tree of inputs together with possible system responses as outputs. One way of dealing with nondeterminism is on-line testing, with random walk being the simplest, but also potentially least efficient such strategy. More sophisticated on-line approaches include, e.g. Nachmanson et al. (2004), where game strategies using Markov decision processes together with some transition coverage algorithm are proposed for intelligently choosing compact input actions that broaden the coverage of nondeterministic tests. This latter approach is the one we used as a basis for equivalenceQuery in our current N^* implementation. In an offline setting, even if the system is nondeterministic,

we can still exploit coverage criteria like path coverage, or transition coverage obtained by the Chinese postman algorithm. The test cases in the resulting — off-line generated — test suite have to be executed several times because of nondeterminism, and the chance of discovering "novel" paths is intrinsic in the system. As we mentioned above, the length of the counterexample affects the efficiency of the algorithm. Our implementation of equivalenceQuery attempts to limit the length of test sequences using a heuristic criterion. In particular, once a stated maximum length has been reached, the system is reset, and the shortest path from the initial state to the last visited state is computed. From this state, the exploration is continued until the test is complete.

3. Experimental analysis

In this section, we show an empirical evaluation of N^* on artificially generated NFMs. Our NFM generator has a number of parameters, including number of states, size of input and output alphabets, and amount of nondeterminism. The output machine is complete (input-enabled), which means that in each state there is at least one transition for each symbol in the input alphabet. The outputs of actions are selected randomly, optionally with a defined probability for each output. The amount of nondeterminism is specified by the number of states that have nondeterministic outgoing transitions, and the maximum number of such transitions. Machines are built by generating the specified number of states, and then adding to each state all possible outgoing transitions. Random output labels are assigned to transitions based on the specified cardinality of the output alphabet. The generator has an option to generate "balanced", rather than fully random machines. Balancing is achieved by keeping track of the input degree of each state, and then trying to keep such degree balanced among all states by randomly selecting destination of transitions from the set of states with lower input degree. In this way, we wish to avoid subsets of nodes in which the average input degree is much higher or much lower than the overall average input degree.

In our experimental setup, N^* must identify NFMs according to two different implementations of outputQuery and equivalenceQuery. The *ideal* implementation corresponds to the theoretical baseline, and it is feasible in practice because we know the structure of the generated NFMs. The *real* implementation corresponds to query approximations, where outputQuery and equivalenceQuery are oblivious of the internal structure of the generated NFMs. We consider families of parametric NFMs $M = (Q, \Sigma_I, \Sigma_O, q_o, \tau)$ where we vary the number of states $|Q|$, the number of input/output symbols $|\Sigma_I| = |\Sigma_O|$, the fraction of states affected by nondeterministic transitions, and the degree of nondeterministic transitions. The settings of parameters p – minimum probability of nondeterministic transitions – and c – required confidence – for the real implementation of outputQuery are $p = 0.5$ and $c = 0.9999$. Notice that $p = 0.5$ corresponds to the actual minimum probability of nondeterministic transitions in artificially generated machines as long as the degree of nondeterministic transitions is always limited to two. The parameter $c = 0.9999$ was sufficient in practice to guarantee a precise approximation of outputQuery in our experiments.[3]

3. Experiments run on an Intel 3.4GHz i7 PC with 32GB of RAM, running 64 bit Microsoft Windows 7. N^* and the real/ideal oracles are implemented in AIDE. Regarding memory, it is worth noticing that our experiments never required more than 3GB of RAM on any single identification task.

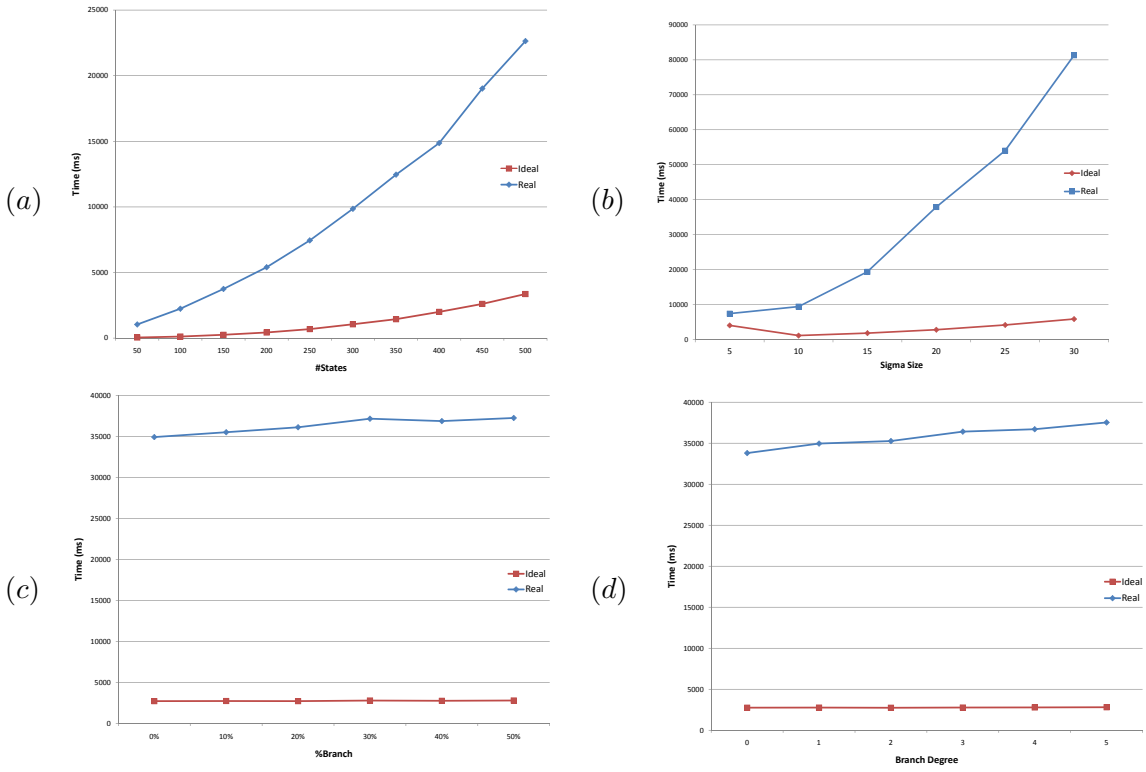


Figure 5: Ideal vs. real implementations of N^*; in plot (a), the x-axis is the number of states $|Q|$; in plot (b), the x-axis is the number of input symbols $|\Sigma_I|$; in plot (c), the x-axis is the percentage of states having a nondeterministic transition; plot (d) presents the case of different branch degrees, from 0 extra transitions in each state to 5 extra ones. In all the plots, diamonds and squares represent performances of N^* using ideal and approximation oracles, respectively; the time scale on the y-axis is linear and reports milliseconds in all the plots.

A summary of the results is presented in Figure 5. In particular, Figure 5 (a) shows the effect of an increasing number of states on the time required to identify the unknown NFM. In these experiments, we have $|\Sigma_I| = |\Sigma_O| = 10$ and nondeterministic transitions with degree two are present in 10% of the states. Here, we can observe the absolute gap between the ideal and the real implementation of N^* growing with the number of states: from 1 second to identify a machine with $|Q| = 50$, to 22 seconds for a machine with $|Q| = 500$. However, if we consider the relative gap between real and ideal implementations, then we see that this is not increasing. The factor between real to ideal settings is 21.6 for $|Q| = 50$ and 6.7 for $|Q| = 500$. This result is to be expected considering that the number of sequences characterized by the same input symbols and different output symbols is growing with the number of states, and so is the size of the observation table. However, the overall time required for approximate identification is reasonable even for $|Q| = 500$ — about 22 CPU seconds. It is important to notice that, in these experiments, the first conjecture supplied to equivalenceQuery results in a match with the unknown NFM, i.e., no counterexample is generated. Therefore, the widening time gap is due to outputQuery being called for an

increasing number of times and responding with heftier and heftier outputs. In particular, with $|Q| = 500$, N^* reaches a solution in 21.6 seconds, filling a table with approximately 5000 rows and 10 columns. The number of columns is $|\Sigma_I|$, since the first conjecture is accepted by equivalenceQuery, and no suffixes are added to E_n in this case.

Figure 5 (*b*) shows the effect of the size of input and output alphabets. The number of states is $|Q| = 300$, and nondeterministic transitions with degree two are present in 10% of the states. Also in this case, we can observe that MAT approximation slows down N^*. Starting from $|\Sigma_I| = 5$, the gap between implementations is growing from a factor of 1.8 to a factor of 13. Notice that for $|\Sigma_I| = 5$, the ideal implementation of N^* performs slightly worse than for $|\Sigma_I| = 10$. In this case, a relatively small alphabet size causes a lot of aliasing in the rows of T_n, which results in calls to equivalenceQuery failing with (potentially) long counterexamples. In turn, this implies that N^* must spend a lot of effort to make the observation table closed again, while the size of such table is also growing. For instance, the number of columns with $|\Sigma_I| = 5$ is more than 15, whereas for $|\Sigma_I| \geq 10$, the number of columns becomes approximately equal to $|\Sigma_I|$. The number of cells are (approximately) 15, $30K$, $68K$, $120K$, $188K$ and $270K$ for $|\Sigma_I| = 5$, 10, 15, 20, 25 and 30, respectively. On the other hand, for $|\Sigma_I| \geq 10$ the number of calls to equivalenceQuery is always one.

Figure 5 (*c*) shows the effect of the percentage of states affected by a nondeterministic transition with $|Q| = 300$ and $|\Sigma_I| = |\Sigma_O| = 20$. The time required by the real implementation of N^* is within a factor of 12.5 with respect to the ideal one. A slight growth in the gap is also noticeable for increasing values of the percentage. The growth seems however pretty slow – less than 2.5 seconds – which, in accordance with the results considered before, supports the conjecture that a branching factor of two in nondeterministic transitions is not making the identification problem excessively hard in practice. We also observe that when the percentage is 0 – leftmost point in Figure 5 (*c*) – the unknown machine is indeed deterministic, but the run time of the real implementation of N^* is not appreciably faster than with actual NFMs. This is because the real implementation of N^* is oblivious of the characteristics of the underlying machines, and thus outputQuery keeps repeating tests as if the target machine was nondeterministic. Figure 5 (*d*) shows the effect of an increasing amount of nondeterminism. Here, we have $|Q| = 300$, $|\Sigma_I| = |\Sigma_O| = 20$, and nondeterminism affects 10% of the states. We change the number of extra (nondeterministic) transitions from 0 (fully deterministic) to 5. In the case of the ideal implementation, this does not cause a decay in performances, whereas in the real implementation, the identification time increases from 33 seconds with a branching factor of 0, to 37 seconds with a branching factor of 5. The reason of such a stark difference between real and ideal is, like in the case of Figure 5 (*c*), that the real implementation needs more trials to come up with the output tree. This is due to the fact that (*i*) accessing a state by its access sequence requires more effort, and (*ii*) because of more nondeterminism, the output tree is likely to have more branches.

Overall, we can conclude that the performances gap between ideal and real implementations of N^* is manageable, as long as the amount of nondeterminism is limited and the number and length of counterexamples to be processed is also relatively small. This is in accordance with the theoretical bounds on complexity, and it gives an idea about the usefulness of N^* as an identification algorithm. In practice, one may ask what happens when parameters p and c are not appropriate, or when the NRE hypothesis is not satisfied by the target machine. Let us assume that there is at least one event, i.e., nondeterministic

Table 1: Effects of p and c settings on N^*: varying the confidence parameter c (left), and varying the minimum transition probability p (right). In both tables, "Time" is the average CPU time spent by N^* (considering successful cases only), "Repetitions" is the number of times a query on $w \in \Sigma_I^*$ is repeated inside outputQuery$(.,.,w)$, and "Success" is the percentage of times in which N^* was successful in a family of 50 artificially generated NFMs.

c	Time(ms)	Repetitions	Success
0.9	-	3	0%
0.99	1875	6	20%
0.999	2054	9	90%
0.9999	2641	13	98%
0.99999	3644	16	100%

p	Time(ms)	Repetitions	Success
0.5	-	13	0%
0.4	6426	18	30%
0.3	6807	25	50%
0.2	11890	41	84%
0.1	19875	87	100%

transition, whose probability is less than p. We call this a *rare-event* and we foresee different scenarios. If the rare-event does not happen during the execution of either outputQuery or equivalenceQuery, N^* infers a model missing the rare event. If the rare event does not happen during the execution of outputQuery, but it does happen during equivalenceQuery and it yields a counterexample. Since this counterexample is not compatible with the observation table, N^* fails. If the rare event happens during the execution of outputQuery, but not enough times to allow a complete answer to be built, then outputQuery fails, and so does N^*. Finally, it is possible that the rare event is not seen in some outputQuery calls but it is consistently seen in others. Usually, this leads to the creation of some states that do not exist in the real system. Afterward, two things may happen: first, after adding the counterexample, the table is still closed which is an obvious sign of missing results from outputQuery. The other is that the table is not closed, the learner tries to make it closed, and this may cause further missing elements. Still in some cases, equivalenceQuery — which is based on edge coverage — is not able to find a counterexample, and thus N^* delivers its (imprecise) conjecture as a result.

Given the scenarios above, accepting the practical possibility that N^* may fail, brings forth the need of further experimentation aimed at understanding this phenomenon from a quantitative point of view. In particular, we investigate two settings: *(i)* A family of NFMs is generated with $|Q| = 300$, $|\Sigma_I| = |\Sigma_O| = 20$, 10% of states affected by nondeterministic transitions, at most two such transitions for each state affected, and the same probability of taking any of the two transitions, i.e., $p = 0.5$. We fix the confidence c to stricter and stricter values to observe whether the chance of failure decreases. And *(ii)* A family of NFMs is generated with $|Q| = 100$, $|\Sigma_I| = |\Sigma_O| = 10$, 20% of states affected by nondeterministic transitions, at most two such transitions for each state affected. The probability of taking any of the two transitions is now unbalanced, thus creating a rare event. In this way we can observe whether the chance of failure decreases when setting p to adequate values.

Table 1 reports the results of the experiments described above. Each row in the tables summarizes the results on a family of 50 artificially generated NFMs. In Table 1 (left) we can observe the behavior of N^* when the confidence parameter c increases. Noticeably, a confidence level such that of $1 - c = 10^{-5}$ ensures a 100% success rate, i.e., all the NFMs in the pool of fifty are identified correctly. At the same time, the average CPU time spent on successful identifications is only doubled. Notice that setting $1 - c \geq 10^{-3}$ is largely insufficient to meet acceptable confidence levels, even when the parameter p is known in

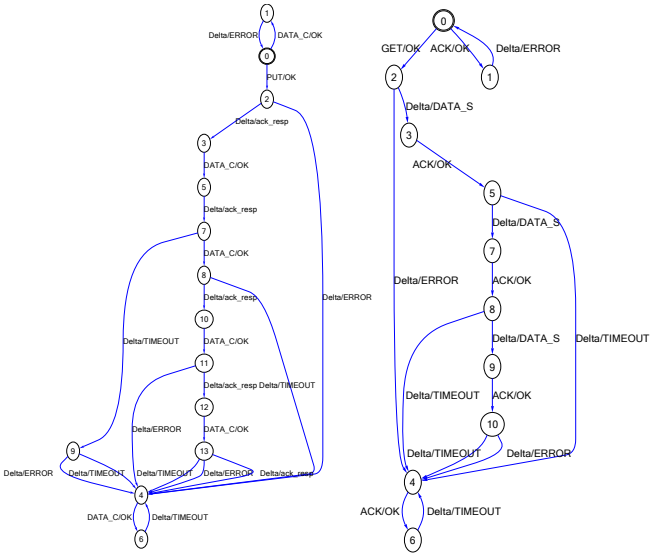
advance or it can be estimated with sufficient accuracy. In Table 1 (right), we report the behavior of N^* on the second experiment, i.e., given a required confidence $c = 0.9999$, we try various settings of p, but the actual value for the target machines is $p = 0.1$. Also in this case, we can observe that N^* is able to perform reasonably only if the right value of p is matched. In practice, the number of repetitions of a query inside outputQuery turns out to be the yardstick of N^* performances, both in terms of time, and in terms of success rate. This is true insofar the probability of rare events is not too small to prevent reasonable approximations by equivalenceQuery. Indeed, in cases where the actual value of p is less than 0.1, we can observe several conjectures which pass the equivalenceQuery test, but are not equivalent to the target machine.

4. Case Study: learning a model for a TFTP server

Trivial File Transfer Protocol (TFTP) is a file transfer protocol which is often used, e.g., for automated transfer of boot and configuration files from a server to diskless workstations in a local area network. TFTP is defined by RFC 1350, and we wish to identify an implementation of the protocol[4] using AIDE — plus an adaptor — as TFTP client. The adaptor facilitates the communication between AIDE and the TFTP server and performs abstraction/concretization steps. Since we are interested in learning the protocol for transfer sessions, we limit the learner to a single transfer in a query, and we consider reading/writing of files smaller than 512 bytes which can be transferred within one packet. Furthermore, we abstract away the file name parameter and the sequence number in data and acknowledgement packets. The adaptor knows a – minimum – set of parameters for these actions, and by receiving one of these (abstract) inputs, it generates a corresponding concrete stimulus with a random parameter. Indeed, this is exactly the source of non-determinism in the observable behavior.

The learned model is presented in Figure 6, showing file upload (left) and file download (right). If we consider the latter, then we see that, in the initial state, the answer of the server to any acknowledgment packet is an error. Once the client sends a GET, the server may reply with ERROR, e.g., if the requested file does not exist, or DATA_S, i.e., the packet with the requested file. Sending an acknowledgment packet after the data request informs the server about the last received packet. So, if the server receives an acknowledgment showing that client did not receive the packet, it re-sends the data packet. This can happen twice, and, at the third time, the server replies with error response. After any ERROR response from the server, the connection is closed and all the requests will be timed-out. Here, the on-line random-walk testing was used for equivalenceQuery. We have also tried the edge-coverage based method, but this method was not able to find a counterexample after the second equivalence query, yielding an inexact model. In order to learn the models in Figure 6 we set $c = 0.9999$ and $p = 0.5$ in outputQuery. With these settings, at least 95% of the experiments performed by N^* are originated by output queries, and the remaining from equivalence ones. In particular, learning the upload model required 322 output and 3 equivalence queries corresponding to more than 200,000 experiments, whereas the download model required 102 output and two equivalence queries corresponding to about

4. We used Open TFTP Server (http://sourceforge.net/projects/tftp-server/), an open source implementation of TFTP on Microsoft Windows.



Action	I/O	Description
GET	I	downloading request
PUT	I	uploading request
ACK_C	I	client acknowledgement
DATA_S	O	packet sent by server
DATA_C	I	data packet sent by client
ERROR	O	get/put failure
ACK_S	O	server acknowledgement
TIMEOUT	O	server timeout
OK	O	input was correct
Delta	I	silent event

Figure 6: Identified partial model of TFTP server related to a file upload (left) and a file download (right).

53,000 experiments. Overall, the time taken to learn the two models was about $300^{minutes}$, of which only a negligible fraction is spent inside N^*, and the most time-consuming parts are network delays and the happened time-outs to investigate output on actions which are not enabled. This explains why N^* is able to learn abstract NFMs with hundreds of states in less than 25s, but it requires around 5^{hours} to identify a TFTP model with only 14 states.

5. Conclusions

Summing up, we have presented three main contributions in the direction of learning nondeterministic Mealy machines. The first one is the algorithm N^* to infer NFMs. N^* is based on an extension of the MAT concept introduced by Angluin to learn finite state automata. We have provided an algorithmic characterization of N^* as well as practical suggestions to implement N^* for applications where the ideal oracle is not available. The second one is an extensive experimental analysis on artificially generated NFMs, a quantitative assessment of the difference in performances between ideal and real implementations of N^*, including the relationship between the parameters of approximation oracles and those of the artificial machines. The third one is the evaluation of N^* to learn the TFTP protocol by interfacing its implementation to an off-the-shelf TFTP server. Finally, even if we do not describe it in this paper, the free open-source framework AIDE has been developed as part of this research, and enabled the experimental analysis herein presented.

The overall result of our experimental analysis is that AIDE/N^* can learn small-to-medium sized systems, and its effectiveness depends critically on the correct estimation of the event probability and the degree of non-determinism. When the estimation is correct, and the system is not heavily non-deterministic, identification could be performed in reasonable time and memory. From a theoretical point of view, this is explained by equation

(2) which contains an exponential factor whose base is the maximum number of nondeterministic transitions occurring in a single state. Experimental results confirm the theoretical bound, since machines with relatively small percentage of states affected by degree-2 nondeterminism can be learned, whereas increasing the percentage of states *and* the degree of nondeterminism can make identification challenging in practice.

References

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

Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. Angluin-style learning of NFA. In *IJCAI*, volume 9, pages 1004–1009, 2009.

Khaled El-Fakih, Roland Groz, Muhammad Naeem Irfan, and Muzammil Shahbaz. Learning finite state models of observable nondeterministic systems in a testing context. In *22nd IFIP International Conference on Testing Software and Systems*, pages 97–102, Natal, Brazil, 2010.

E Mark Gold. System identification via state characterization. *Automatica*, 8(5):621–636, 1972.

Falk M. Howar. *Active learning of interface programs*. PhD thesis, Technischen Universitat Dortmund, 2012.

Muhammad Naeem Irfan. *Analysis and optimization of software model inference algorithms*. PhD thesis, Universita de Grenoble, Grenoble, France, September 2012.

David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

Maik Merten, Falk Howar, Bernhard Steffen, and Tiziana Margaria. Automata learning with on-the-fly direct hypothesis construction. *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 248–260, 2012.

Lev Nachmanson, Margus Veanes, Wolfram Schulte, Nikolai Tillmann, and Wolfgang Grieskamp. Optimal strategies for testing nondeterministic systems. *ACM SIGSOFT Software Engineering Notes*, 29(4):55–64, 2004.

Oliver Niese. *An integrated approach to testing complex systems*. PhD thesis, Universität Dortmund, Dortmund, Germany, December 2003.

H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(5):393–407, 2009.

M. Shahbaz. *Reverse Engineering Enhanced State Models of Black Box Software Components to Support Integration Testing*. PhD thesis, Institut Polytechnique de Grenoble, Grenoble, France, 2008.

Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

Neil Walkinshaw, Bernard Lambeau, Christophe Damas, Kirill Bogdanov, and Pierre Dupont. STAMINA: a competition to encourage the development and assessment of software model inference techniques. *Empirical software engineering*, 18(4):791–824, 2013.