2021 Seventh Indian Control Conference (ICC)
IIT Mumbai, India. December 20-22, 2021

# Learning event-driven switched linear systems

Atreyee Kundu and Pavithra Prabhakar

*Abstract*— **We propose an automata theoretic learning algorithm for the identification of black-box switched linear systems whose switching logics are event-driven. A switched system is expressed by a deterministic finite automaton (FA) whose node labels are the subsystem matrices. With information about the dimensions of the matrices and the set of events, and with access to two oracles, that can simulate the system on a given input, and provide counter-examples when given an incorrect hypothesis automaton, we provide an algorithm that outputs the unknown FA. Our algorithm first uses the oracle to obtain the node labels of the system run on a given input sequence of events, and then extends Angluin's $L^*$-algorithm to determine the FA that accepts the language of the given FA. We demonstrate our learning algorithm on a numerical example.**

## I. Introduction

Cyber-physical systems that consist of software controlled physical systems have transformed today's transportation, energy and healthcare sectors. Rigorous analysis of these systems has become inevitable given the safety critical environments in which they are deployed. Formal analysis requires a formal model of the system to be analyzed. Often, a model of the system is unavailable, due to, for instance, unknown dynamics or proprietary software, or a complex model maybe available, which is unamenable to analysis. In either case, it is necessary to have techniques to learn such models, from minimalistic knowledge of the system, and some basic operations that are feasible as in a black box setting. In this paper, we investigate the learning problem for certain subclasses of models for cyber-physical systems, wherein, the digital logic (cyber part) is captured as a event-driven deterministic finite state automaton, and the physical system is captured using discrete-time linear dynamics.

In general, switched systems consist of a finite set of subsystems governed by a time-varying switching signal [8, §1.1.2]. We focus on linear subsystems and sets of switching logic that are event-driven in the sense that the active subsystem at any time instant depends on the active subsystem at the previous time instant and the event that was carried out at that time instant. Such switched systems arise naturally in, for example, the setting of a robot-aided neurosurgery [3]. Consider, for instance, a robot that has five modes of operation: (i) Homing, (ii) Autonomous, (iii) Hands-on, (iv) Tele-operation, and (v) Steady. In each mode, it has a certain dynamics, and the mode can change based on certain events. The surgeon is provided with a GUI interface where she can perform an event by pressing a button or touching the robot. Based on the current mode of operation of the robot (subsystem) and the event carried out by the surgeon, the next mode of operation of the robot (subsystem) is selected. To have successful coordination between the human and the robot, it is imperative to understand the functioning of the robot. Hence, we are interested in developing system identification techniques for event-driven switched linear systems by employing automata theoretic learning techniques.Note that we can assume that we have the ability to stimulate the robot with input event sequences, and observe its behavior (execution). We are interested in an algorithm that allows one to compute a "hypothesis" switched system based on such observations on appropriate input event sequences. In addition, if we have the ability to check if the hypothesized system is correct, and obtain a counter-example execution in case it is incorrect, such an algorithm can learn the correct system in finite time.

In general, system identification techniques for switched systems are widely studied, see e.g., [5], [9], [4] for detailed surveys. The problem is known to be NP-hard [7] and is typically performed by collecting input (possibly controlled) - output (possibly noisy) data during the operations of the system. The available techniques can be classified broadly into two categories: offline methods and online methods. In case of the former, access to all data at once is assumed, while in case of the latter, data are available in a streaming (online) fashion. The offline methods include: algebraic method, mixed integer programming method, clustering method, Bayesian method, bounded error identification method, and sparse optimization method. The online methods receive data at each time step and perform two tasks: identification of the subsystem whose dynamics is being followed at that time step and updation of the estimates of the subsystems parameters. In this paper we consider a paradigm shift and explore active learning techniques for system identification of switched systems.

We express an event-driven switched system as an event-deterministic finite automaton (FA), whose node labels are the subsystem matrices. The execution of a switched system depends on an initial (continuous) state and a sequence of input events, and consists of the sequence of states obtain by applying the discrete-time linear dynamics associated with (discrete) state labels that are encountered along the path in the finite automaton induced by the input event sequence. We assume that the set of events that causes switches between

Atreyee Kundu is with the Department of Electrical Engineering, Indian Institute of Science Bangalore, India, E-mail: atreyeek@iisc.ac.in. Pavithra Prabhakar is with the Department of Computer Science, Kansas State University, USA, E-mail: pprabhakar@ksu.edu.

Atreyee Kundu is supported by INSPIRE Faculty Award IFA17-ENG225 by the Department of Science and Technology, Govt. of India. Pavithra Prabhakar was partially supported by NSF CAREER Grant No. 1552668, NSF Grant No. 2008957, ONR YIP Grant No. N000141712577 and USDA Grant No. 2017-67007-26153.

2021 Seventh Indian Control Conference (ICC) | 978-1-6654-0978-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICC54714.2021.9703130

978-1-6654-0978-0/21/$31.00 ©2021 IEEE


Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on May 12,2023 at 05:18:02 UTC from IEEE Xplore. Restrictions apply.

the subsystems and the dimension of the subsystems matrices are known to the Learner. In addition, she has access to two Oracles: (a) An IO-generator, which given an initial state and an input (sequence of events) outputs the execution. Such a IO-generator is typically available for any black box for which an input can be provided and output observed. (b) An Equivalence Checker, which given a hypothesis finite automaton, checks if the language of the hypothesized finite automaton is the same as that underlying the black box automaton, and if they are not equal, provides an input on which the two automata have different outputs. While such an equivalence checker/counter-example generator might be challenging to build, it might still be possible to generate counter-examples by running multiple IO-generator queries and observing the output. Under the above assumptions, our learning algorithm has the following phases: First, the Learner performs a constant number of IO-generator queries to obtain the matrix labeling the last node of the automaton path corresponding to an input, referred to as the *Output*. The switched system learning problem is then reduced to a finite automaton learning problem with *multiple* labels. We provide an extension of Angluin's $L^*$-algorithm [1] to the multiple labels setting, using the notion of *Output* as our observation. The key insight of our algorithm is that we are able to separate the learning tasks into a dynamics identification task and an automata learning tasks. We are able to provide guarantees that our learning algorithm terminates in bounded time and outputs a correct language equivalent switched system. Our algorithm is tested on a numerical example.

The remainder of this paper is organized as follows: In §II we formulate the problem under consideration. Our results appear in §III. We also discuss various features of our learning algorithm in this section. A numerical example is presented in §IV. We conclude in §V with a brief mention of future research directions. Owing to space limitations, we omit proofs of our results and refer the reader to a longer version [6] for the same.

**Notation**. $\mathbb{R}$ will denote the set of real numbers, $I_d$ the $d$-dimensional identity matrix and $I_d^k$ its $k$-th column. For a finite set $A$, its cardinality is denoted by $|A|$. A (finite) sequence over a set $A$ is denoted by listing elements from $A$, e.g., $w = a_1 a_2 \cdots a_n$, where $a_i \in A$, $i = 1, 2, \ldots, n$. $\varepsilon$ denotes an empty sequence. We employ *Last*$(w)$ to denote the last element of the sequence $w$, i.e., *Last*$(w) = w_n$. Also, $w[i \cdots j]$ represents the sequence $a_i \cdots a_j$. Let $A^*$ denote the set of all finite sequences over $A$.

## II. Problem statement

We first define a finite automaton and its language.

*Definition 1:* An event-deterministic labelled finite automaton (FA) is a tuple $\mathcal{D} = (Q, q_0, \Sigma, \Lambda, \delta, \gamma)$, where $Q$ is the set of nodes or (discrete) states, $q_0 \in Q$ is the initial node, $\Sigma$ is a set of events, $\Lambda$ is a set of node labels, $\delta : Q \times \Sigma \to Q$ is the node transition function, and $\gamma : Q \to \Lambda$ is the node labelling function.

In the sequel, we will refer to an event-deterministic labelled finite automaton, as just a finite automaton. The components of a finite automaton will be identified by using subscripts indicating the automaton, such as, $\mathcal{D}$ will refer to the nodes of automaton $Q_{\mathcal{D}}$. When the automaton is clear from the context, the subscripts will be dropped. Note that the transition function of our automaton is deterministic. We will refer to a sequence of event, that is, an element of $\Sigma^*$, as an input (word or sequence). We overload $\delta$ to also denote the function $\delta : Q \times \Sigma^* \to Q$ that given a state and an input word and outputs the state reached on taking the sequence of transitions corresponding to the input word, and is inductively defined as $\delta(q, \varepsilon) = q$ and $\delta(q, ua) = \delta(\delta(q, u), a)$ for all $u \in \Sigma^*$ and $a \in \Sigma$. Similarly, we overload $\gamma$ to a function $\gamma : Q^* \to \Lambda^*$ given by $\gamma(q_0 q_1 \ldots q_n) = \gamma(q_0)\gamma(q_1)\ldots\gamma(q_n)$ for all $q \in Q^*$. We will define the semantics of an FA as a mapping from input words to corresponding sequence of state labels generated by them. We will refer to this mapping as a "language".

*Definition 2:* Given $w = e_1 e_2 \cdots e_n \in \Sigma^*$, *run of $w$ on $\mathcal{D}$* is given by $Run_{\mathcal{D}}(w) = q_0 q_1 \cdots q_n$ for any $w \in \Sigma^*$, where $q_{i+1} = \delta(q_i, e_{i+1})$, for $i = 0, 1, \ldots, n-1$.

*Definition 3:* The *language of $\mathcal{D}$* is a function $L_{\mathcal{D}} : \Sigma^* \to \Lambda^*$ given by $L_{\mathcal{D}}(w) = \gamma(Run_{\mathcal{D}}(w))$.

In the sequel, for learning, we will need the label of the last node reached on reading a word. We will refer to this as the output. This is a generalization of the notion of acceptance of a word by a traditional deterministic finite automaton, where the labels are "final" and "non-final".

*Definition 4:* Given $w = e_1 e_2 \ldots e_n \in \Sigma^*$, the *output of $w$ in $\mathcal{D}$* is the label of the last node of *Run*$(w)$. More specifically, *Output*$_{\mathcal{D}}(w) =$ *Last*$(L_{\mathcal{D}}(w))$.

We consider switched systems consisting of a finite number of discrete-time dynamical systems, each of which is specified by a matrix $A_i$, with the intended dynamics being $x(t+1) = A_i x(t)$, and a switching logic specified using a finite automaton. We capture the switched system holistically as a finite automaton with the matrices being the node labels.

*Definition 5:* A *switched system* is a FA $\mathcal{D}$, whose set of node labels, $\Lambda_{\mathcal{D}}$, is an indexed set of matrices of dimension $d$ represented as $\Lambda_{\mathcal{D}} = \{A_j\}_{j=1}^N$, where $A_j \in \mathbb{R}^{d \times d}$ for every $j$.

In the sequel, we will occasionally refer to the elements of the set $\{A_j\}_{j=1}^N$ as subsystem matrices. An execution of $\mathcal{D}$ from an initial (continuous) state $x \in \mathbb{R}^d$ on an input sequence of events $w$, denoted *Exec*$_{\mathcal{D}}(x, w)$, is the sequence of states reached by applying the dynamics represented by the matrices labelling the nodes in the run of the finite automaton on the input sequence. In the sequel, we will need a general definition of executions from a finite number of, say, $k$ initial states, stored as a $d \times k$-dimensional matrix, each of whose columns represents a state. The execution will be a sequence of $d \times k$-dimensional matrices, where the $i$-th column of these matrices represents the execution starting from the $i$-th column of the initial matrix.

*Definition 6:* An *execution of $\mathcal{D}$*, on a state matrix $X \in \mathbb{R}^{d \times d}$, and a sequence of events, $w = e_1 e_2 \cdots e_n \in \Sigma^*$, is given by *Exec*$_{\mathcal{D}}(X, w) = X_0 X_1 \cdots X_{n+1}$, where $X_0 = X$, $X_{i+1} = \overline{A}_i X_i$, $i = 0, 1, \ldots, n$, and $L_{\mathcal{D}}(w) = \overline{A}_0 \overline{A}_1 \ldots \overline{A}_n$.



Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on May 12,2023 at 05:18:02 UTC from IEEE Xplore. Restrictions apply.

Note that given a state $x \in \mathbb{R}^d$, $Exec_{\mathcal{D}}(x, w)$ denotes the execution from a $d \times 1$ matrix. We use states to refer to both elements of $\mathbb{R}^d$, which are continuous states, and nodes in $Q$, which are discrete states. When there is ambiguity, we will use the prefix "discrete"/"continuous". As before, when the finite automaton or the switched system is clear from the context, we will drop the subscript $\mathcal{D}$ from *Output*, *Run*, $L$ and *Exec*.

Our broad objective is to learn a switched system, which is provided as a black box system.

*Problem 1:* Consider a switched system $\mathcal{D} = (Q_{\mathcal{D}}, q_{0,\mathcal{D}}, \Sigma, \Lambda_{\mathcal{D}}, \delta_{\mathcal{D}}, \gamma_{\mathcal{D}})$. Suppose that we know the set of events, $\Sigma$, and the dimension, $d$, of the elements of $\Lambda_{\mathcal{D}}$. In addition, we have access to two oracles that can perform the following tasks: (a) IO-generator: Given input $(x, w) \in \mathbb{R}^d \times \Sigma^*$, the IO-generator outputs $Exec_{\mathcal{D}}(x, w)$. Note that we can find $Exec_{\mathcal{D}}(X, w)$ for any $d \times k$-matrix by $k$ calls to the IO-generator. (b) Equivalence-checker (counter-example generator): Given a (hypothesis) FA $\mathcal{D}' = (Q', q_0', \Sigma, \Lambda', \delta', \gamma')$ as input, the equivalence checker checks the equivalence of the languages of $\mathcal{D}$ and $\mathcal{D}'$, that is, it outputs if $L_{\mathcal{D}} = L_{\mathcal{D}'}$. If not, then it also outputs a (counter-example) $w \in \Sigma^*$ such that $Output_{\mathcal{D}}(w) \neq Output_{\mathcal{D}'}(w)$. Our objective is to design an algorithm that uses the above oracles to output an automaton $\mathcal{D}'$ such that $L_{\mathcal{D}} = L_{\mathcal{D}'}$.

In the sequel, we will also refer to a call to IO-generator on an input word and a continuous state for obtaining an execution of the black box switched system, as an *observation query*, and the call to the equivalence checker with a hypothesis automaton, an *equivalence query*. Towards solving Problem 1, we will assume that the matrices $\{A_j\}_{j=1}^N$ are full-rank, and devise a learning algorithm that relies on the principles of Angluin's $L^*$ algorithm. Our solution approach broadly consists of the following steps: (i) We use the IO-generator to compute $Output_{\mathcal{D}}(w)$ for a given $w$, thereby reducing the learning problem to that of learning an event-deterministic labelled finite automaton. (ii) We extend the $L^*$-algorithm for deterministic finite automata (with two labels, namely, final and non-final) to the setting of learning event-deterministic finite automata with potentially multiple labels.

## III. SWITCHED SYSTEM LEARNING ALGORITHM

We begin with an algorithm to compute $Output_{\mathcal{D}}(w)$ for a given $w$ by making a sequence of IO-generator queries that provide $Exec_{\mathcal{D}}(x, w')$ as output for a given initial state $x$ and input $w'$. Then we provide an algorithm that learns the underlying finite automaton that has access to the equivalence checker and the algorithm for computing $Output_{\mathcal{D}}(\cdot)$.

The computation of $Output_{\mathcal{D}}$ relies on the fact that a matrix $A$ can be uniquely computed given a set of basis vectors and their transformation on the application of $A$, when $A$ is full-rank. Let GetMatrix be a function that takes as input a matrix $X$ whose columns form a basis, and the transformation of those vectors on a matrix $A$, given by a matrix $X' = AX$, and returns $A$. More precisely, GetMatrix$(X, X')$ takes as input two matrices $X, X' \in \mathbb{R}^{d\times d}$ whose columns form a basis, and solves the systems of linear equations $AX = X'$ for $A \in \mathbb{R}^{d\times d}$, and returns $A$. Such a matrix can be constructed effectively by solving the system of linear equations, and the uniqueness of the solution is guaranteed by well-known results from linear algebra.

To obtain $Output_{\mathcal{D}}(w)$, we need to find two sets of basis vectors, where the second one corresponds to a transformation of the first using the matrix $Last(\gamma_{\mathcal{D}}(Run_{\mathcal{D}}(w)))$. The algorithm is quite straight forward. Consider $I_d$, a $d \times d$ identity matrix, whose columns form a basis. Let $X = Last(\text{Exec}_{\mathcal{D}}(I_d, w[1 \cdots n-1]))$, where $w$ is a sequence of $n$ events. Note that the columns of matrix $X$ also form a basis, because all the matrices in $\gamma_{\mathcal{D}}(Run_{\mathcal{D}}(w))$ are full rank matrices. Similarly, let $X' = Last(\text{Exec}_{\mathcal{D}}(I_d, w[1 \cdots n]))$, which again represents a basis. Moreover, $X' = Output_{\mathcal{D}}(w)X$. Hence, $Output_{\mathcal{D}}(w)$ is given by GetMatrix$(X, X')$. This construction of $Output_{\mathcal{D}}(w)$ is outlined in Algorithm 1.

**Algorithm 1** Computation of $Output_{\mathcal{D}}(w)$

**Input:** The dimension of the subsystems matrices, $d$ and a sequence of events, $w \in \Sigma^*$.
**Output:** $Output_{\mathcal{D}}(w)$.
1: **if** $w = \varepsilon$ **then**
2: Output $\text{Exec}_{\mathcal{D}}(I_d, \varepsilon)$ and terminate.
3: **else**
4: Set $X = Last(\text{Exec}_{\mathcal{D}}(I_d, w[1 \cdots n-1]))$
5: Set $X' = Last(\text{Exec}_{\mathcal{D}}(I_d, w[1 \cdots n]))$
6: Output GetMatrix$(X, X')$ and terminate.
7: **end if**

*Lemma 1:* Given $w \in \Sigma^*$, Algorithm 1 outputs $Output_{\mathcal{D}}(w)$.

Armed with Algorithm 1, we proceed towards extending $L^*$-algorithm from the learning literature to the learning of a FA $\mathcal{D}^*$ that accepts the language of $\mathcal{D}$.

Let us fix an unknown finite automaton $\mathcal{D}$, for which we know the set of events $\Sigma$ and the dimension of the matrices in $\Lambda_{\mathcal{D}}$. Our objective is to output a finite automaton $\mathcal{D}'$ such that $L_{\mathcal{D}} = L_{\mathcal{D}'}$. We have access to an algorithm for computing $Output_{\mathcal{D}}(w)$ for any given input $w$, from Algorithm 1.

The broad framework of our learning approach based on Angluin's $L^*$ algorithm is as follows: at each step of the learning algorithm, the Learner maintains two sets of input words (sequences over $\Sigma$): $Q$, a set of access words, and $T$, a set of test words. Intuitively, the set $Q$ represents a set of input words that reach distinct states in any minimal finite automaton $\mathcal{D}^*$ representing the language to be learnt. Note that for any two distinct states of $\mathcal{D}^*$, there is an input word, that will distinguish the behaviors from those states. $T$ is a finite set of input words that can distinguish any pair of states in $Q$. This property is referred to as $(Q, T)$ being $\mathcal{D}$-separable. The algorithm consists of judiciously expanding $Q$ and when required $T$, so that $(Q, T)$ separability is maintained. In each step, a hypothesis automaton is constructed from $Q$ by possibly adding states to "close" the automaton, that is, to ensure that there is a next state on every event from every



Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on May 12,2023 at 05:18:02 UTC from IEEE Xplore. Restrictions apply.

state. The language of the closed automaton is compared with $\mathcal{D}$ using an equivalence query, and a counter-example if returned, is used to identify a state that has not been captured by the set $Q$ and added. The process is repeated until a finite automaton which passes the equivalence query is found.

First, we define when two input words are equivalent with respect to a set of test words $T$.

*Definition 7:* Given a set $T \subseteq \Sigma^*$, and two words $u, v \in \Sigma^*$, we say that $u, v$ are *T-equivalent* with respect to $\mathcal{D}$, denoted by $u \equiv_T^{\mathcal{D}} v$, if $\textit{Output}_{\mathcal{D}}(uw) = \textit{Output}_{\mathcal{D}}(vw)$ for all $w \in T$.

Given a finite $T$ and input words $u, v$, we can algorithmically check if $u, v$ are $T$-equivalent, by iterating over words $w \in T$ and using Algorithm 1 to check if $\textit{Output}_{\mathcal{D}}(uw) = \textit{Output}_{\mathcal{D}}(vw)$. Note that if $u, v$ are not $T$-equivalent, then some word $w$ from $T$ distinguishes them, in terms of the label of the last state reached after reading $w$ from the states reached after reading $u$ and $v$, respectively. This leads us to the notion of separability, which guarantees that the states reached by words in a set $Q$ of access strings are distinct, using a finite set of test strings $T$ that witness the distinguishability.

*Definition 8:* The pair $(Q, T)$ is called $\mathcal{D}$-*separable*, if no two distinct words in $Q$ are $T$-equivalent with respect to $\mathcal{D}$.

Again, given that we can check if $u, v$ are $T$-equivalent for a finite $T$, we can also algorithmically check if $(Q, T)$ $\mathcal{D}$-separable, when $Q$ is also finite. Given a set of access strings $Q$ that reach distinct states of $\mathcal{D}$, we want to hypothesize a finite automaton that captures the language of $\mathcal{D}$. We need to identify the states and transitions of this automaton. We can consider $Q$ to represent the states, with the interpretation that they represent the states reached in $\mathcal{D}$ when given themselves as input. In order to define the edge, for every $q \in Q$ and $e \in \Sigma$, we need to identify a word in $Q$ that corresponds to $qe$. We can search for a word in $Q$, that is $T$-equivalent to $qe$. Note that there is at most one such word in $Q$ if $(Q, T)$ is separable. However, no such word might exist. Hence, we add those words to $Q$, until $Q$ is closed with respect to the "next step" operation. Next, we formalize the notion of closure, and the hypothesis automaton constructed when a closed pair $(Q, T)$ is given.

*Definition 9:* The pair $(Q, T)$ is called $\mathcal{D}$-*closed*, if for every $q \in Q$ and $e \in \Sigma$, there exists $q' \in Q$ such that $qe \equiv_T^{\mathcal{D}} q'$.

*Definition 10:* Consider a $\mathcal{D}$-separable and $\mathcal{D}$-closed pair $(Q, T)$. A hypothesis automaton for $(Q, T)$ is a finite automaton $\mathcal{D}' = (Q', q_0', \Sigma, \Lambda', \delta', \gamma')$, where $Q' = Q$ with the empty sequence of events, $\varepsilon$, being the initial node, that is, $q_0' = \varepsilon$, $\Lambda' = \{\textit{Output}_{\mathcal{D}}(q) \,|\, q \in Q\}$, for any $q, e$, $\delta(q, e) = q'$, where $q' \in Q$ is such that $qe \equiv_T^{\mathcal{D}} q'$, and for any $q$, $\gamma(q) = \textit{Output}_{\mathcal{D}}(q)$.

Note that our definition of hypothesis automaton is well-defined, since, in the definition of $\delta'$, there is at most one $q' \in Q$ satisfying $qe \equiv_T^{\mathcal{D}} q'$, because of the separability property of $(Q, T)$. Also, checking for whether a pair of finite sets $(Q, T)$ is closed and the construction of the hypothesis automaton for $(Q, T)$ are computable.

Our learning algorithm is summarized in Algorithm 2. The details and correctness of the algorithm depend on the following results.

**Algorithm 2** Learning a minimal FA whose semantics is $L_{\mathcal{D}}$

**Input:** The set of events, $\Sigma$ and the dimension of the subsystems, $d$, Algorithm for computing $\textit{Output}_{\mathcal{D}}$ and Counter-example generator for the language $L_{\mathcal{D}}$
**Output:** A FA $\mathcal{D}'$ whose language is $L_{\mathcal{D}}$

1: Set $Q = T = \{\varepsilon\}$.
2: Apply Lemma 4 to find $\tilde{Q} \supseteq Q$ such that $(\tilde{Q}, T)$ is $\mathcal{D}$-separable and $\mathcal{D}$-closed.
3: Set $Q = \tilde{Q}$
4: Construct a hypothesis automaton, $\mathcal{D}'$ for the pair $(Q, T)$
5: Check equivalence of $\mathcal{D}'$ and $\mathcal{D}$
6: **if** a counter-example $w \in \Sigma^*$ is returned **then**
7: Apply Lemma 5 to expand $Q$ and $T$ towards obtaining a $\mathcal{D}$-separable pair $(\tilde{Q}, \tilde{T})$
8: Set $Q = \tilde{Q}$ and $T = \tilde{T}$
9: Go to Line 2
10: **else**
11: Output $\mathcal{D}'$ and terminate.
12: **end if**

First, we show that there is an upper-bound on the size of $Q$ for any $(Q, T)$ pair that is $\mathcal{D}$-separable. Intuitively, since, each access string in $Q$, necessarily reaches a different state in any minimal finite automaton for $L_{\mathcal{D}}$, due to the fact that some string (from $T$) distinguishes it from any other string in $Q$, the size of $Q$ can be at most the number of states of a minimal finite automaton, which is less than $N_{\mathcal{D}}$, the number of states of $\mathcal{D}$.

*Lemma 2:* If the pair $(Q, T)$ is $\mathcal{D}$-separable, then $|Q|$ is at most $N_{\mathcal{D}}$.

The next result states that if $(Q, T)$ is not closed, then $Q$ can be expanded, while keeping $T$ and the $\mathcal{D}$-separability of $(Q, T)$ intact. Note, however, that from Lemma 2, there is an upper bound on the size of $Q$, so, the next Lemma also implies that by expanding $Q$ at most $N_{\mathcal{D}}$ times, we can obtain a pair $(Q, T)$, that is both closed and separable. Also, the expansion at each step is computable. Line 2 of Algorithm 2 uses this to compute a closed and separable pair $(Q, T)$.

*Lemma 3:* If the pair $(Q, T)$ is $\mathcal{D}$-separable but not $\mathcal{D}$-closed, then there is a $q \in Q$ and $e \in \Sigma$ such that $(Q \cup \{qe\}, T)$ is $\mathcal{D}$-separable.

*Lemma 4:* For every $\mathcal{D}$-separable pair $(Q, T)$, we can compute a $\mathcal{D}$-closed and $\mathcal{D}$-separable pair $(\tilde{Q}, T)$, where $Q \subseteq \tilde{Q}$, in time at most $O(N_{\mathcal{D}})$

Next, we present the details of the algorithm for expanding $(Q, T)$ if the hypothesis automaton is incorrect. We will use a counter-example returned by the equivalence checker to expand the pair $(Q, T)$ such that separability is still maintained. This will be again followed by a closure operation to obtain the next hypothesis automaton, and the loop will continue until a hypothesis automaton whose language is that of $\mathcal{D}$ is found.



Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on May 12,2023 at 05:18:02 UTC from IEEE Xplore. Restrictions apply.

*Definition 11:* A counter-example for $\mathcal{D}'$ with respect to $\mathcal{D}$ is an input word $w \in \Sigma^*$ for which the languages of the two automata have different outputs, that is, $\mathit{Output}_{\mathcal{D}}(w) \neq \mathit{Output}_{\mathcal{D}'}(w)$.

*Lemma 5:* Suppose that the pair $(Q, T)$ is $\mathcal{D}$-separable and $\mathcal{D}$-closed, and $\mathcal{D}'$ be the corresponding hypothesis FA. Given a counter-example $w$ for $\mathcal{D}'$ with respect to $\mathcal{D}$, we can compute $q \in \Sigma^* \setminus Q$ and $t \in \Sigma^*$ such that the pair $(Q \cup \{q\}, T \cup \{t\})$ is $\mathcal{D}$-separable using at most $O(\log(|w|))$ IO-generator queries.

Next, we state the correctness of the finite automaton learning algorithm.

*Theorem 1:* Algorithm 2 always terminates and outputs a finite automaton whose language is $L_{\mathcal{D}}$.

*Remark 1:* Our algorithm is similar to Angluin's algorithm, however, the technical development is performed using the notion of *Output* that generalizes two labels to multiple labels, and *Output* can be computed using IO-generator queries for our subclass of linear switched systems.

To wrap up, let us discuss the problem of learning the switched system. Given a switched system $\mathcal{D}$ with $d$ and $\Sigma$ known, Algorithm 2 outputs a switched system $\mathcal{D}'$ whose executions coincide with that of $\mathcal{D}$.

*Corollary 1:* Algorithm 2 outputs a switched system $\mathcal{D}'$ such that $\mathit{Exec}_{\mathcal{D}}(x, w) = \mathit{Exec}_{\mathcal{D}'}(x, w)$ for every $x \in \mathbb{R}^d$ and $w \in \Sigma^*$.

*Remark 2:* Given the set of events, $\Sigma$, the dimension of the subsystems matrices, $d$, and the IO-generator and counter-example generators, Algorithm 2 learns an FA that accepts the semantics of the underlying FA of the unknown switched system under consideration. The learning technique employed in Algorithm 2 is an extension of the $L^*$-algorithm. In the $L^*$-algorithm, the Learner learns an event-deterministic unlabelled finite automaton that accepts a certain language $L$, with the aid of an Oracle called the *minimally adequate teacher* (MAT). An automaton $\mathcal{A}$ under consideration in [1] is a tuple $(P, p_0, \Gamma, F, \mu)$, where $P$ is a finite set of nodes, $p_0 \in P$ is the unique initial node, $\Gamma$ is a finite set of alphabets, $F \subseteq P$ is a finite set of accepting (or final) nodes, and $\mu : P \times \Gamma \to P$ is the node transition function. The language of $\mathcal{A}$ is the set of all finite words (strings of alphabets) such that the automaton reaches a final node on reading them, i.e., a word $w = w_1 w_2 \cdots w_m$, $w_k \in \Gamma$, $k = 1, 2, \ldots, m$, belongs to the language of $\mathcal{A}$, if $\mu(\cdots(\mu(\mu(p_0, w_1), w_2), \cdots, w_m) \in F$. The MAT knows $L$ and answers two types of queries by the Learner: *membership queries*, i.e., whether or not a given word belongs to $L$, and *equivalence queries*, i.e., whether a hypothesis automaton specified by the Learner is correct or not. If the language of the hypothesis automaton differs from $L$, then the MAT responds to an equivalence query with a counter-example, which is a word that is misclassified by the hypothesis automaton. The class of automata considered in this paper differs structurally from the class of automata considered in [1] in the following ways: (a) $\mathcal{D}$ has 0-many accepting nodes, and (b) the nodes of $\mathcal{D}$ are labelled with matrices. In Algorithm 2 we modify the $L^*$-algorithm to cater to learning of $\mathcal{D}$. At this point, it is important to highlight that throughout this paper we have employed notations, terminologies and concepts from the version of $L^*$-algorithm presented in [10]. Loosely speaking, the IO-generator and counter-example generator together play the role of a MAT. Indeed, the IP-generator provide the Learner with finite traces of state trajectories of the unknown $\mathcal{D}$ under consideration. The Learner then uses this information to compute $\mathit{Output}_{\mathcal{D}}(w)$ for $w \in \Sigma^*$ that satisfy certain conditions. In addition, the the counter-example generator facilitates checking correctness of an FA hypothesized by the Learner.

*Remark 3:* Earlier in [2] the role of labels on the nodes of an automaton were employed in the setting of the $L^*$-algorithm to aid the learning process. The authors allow the MAT to make an automaton easier to learn by adding binary scalar labels to its nodes, either carefully or randomly chosen. When the Learner performs a membership query for a string, then she not only receives whether it is accepting or not, but also is provided with the label of the node that the automaton reaches on its application. It is shown that if the node labels are distinct, then the learning process becomes easier, and if all the node labels are same, then the learning may require an exponential number of queries. The above set of observations does not extend readily to our setting due to the structural difference of our FA's with the class of automata considered in the $L^*$-algorithm described above. Indeed, our FA's do not have final nodes and labelling of the nodes with matrices is governed by the underlying switching rules of the system under consideration. Beyond identification of switched systems, our learning algorithm is applicable to the general setting of learning deterministic finite automaton with 0-many final nodes and all nodes labelled with full-rank matrices.

*Remark 4:* Notice that the IO-generator and the counter-example generator can be thought of as a simulation model of the unknown switched system, $\mathcal{D}$. In modern industrial setups, simulation is of prior importance. Such models for complex systems are often provided by the system manufacturers. The mathematical models of the system components and the constraints on their operations underlying the simulation model are typically not made known explicitly to the user, but the model can be used to study the system behaviour with respect to various sets of inputs prior to their application to the actual system. Given a simulation model that allows the set of operations by the user required for our setting, the Learner can generate finite traces of trajectories of a switched system with respect to sets of initial states and sequences of events. This serves for the purpose of Algorithm 1. For the generation of a counter-example, the Learner can apply sequences of events of increasing length (up to a sufficiently large number) and match the labels of the nodes reached on $\mathcal{D}$ and $\mathcal{D}'$.

## IV. A NUMERICAL EXAMPLE

We construct a MATLAB (R2020a) routine `fa-oracle.m` that knows $\mathcal{D}$ and can perform the



Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on May 12,2023 at 05:18:02 UTC from IEEE Xplore. Restrictions apply.

following task: accept an input $(x,w)$ and output $\mathit{Exec}_{\mathcal{D}}(x,w)$. Our Learner routine `fa-learn.m` uses `fa-oracle.m` as both an IO-generator and a Counter-example generator. Using `fa-oracle.m` as an IO-generator is straightforward. Towards using it as a Counter-example generator, `fa-learn.m` performs the following tasks: (a) it fixes a hypothesis automaton $\mathcal{D}'$, (b) chooses a large number $L$, (c) computes $\mathit{Output}_{\mathcal{D}}(w)$ for all possible $w$ of increasing length, one at a time, by means of Algorithm 1 and the routine `fa-oracle.m`, and (d) matches $\mathit{Output}_{\mathcal{D}}(w)$ with $\mathit{Output}_{\mathcal{D}'}(w)$. This procedure is continued until either a counter-example $w$ is obtained or all $w$ of length $i = 1, 2, \ldots, L$ are exhausted.

We consider a linear plant with 3 modes of operations. Under a healthy condition, the plant follows a pre-specified schedule for mode selection. Whenever a fault occurs, the plant continues to dwell on the current mode of operation until the fault is cleared. This setting can be expressed as an internally event-driven switched system for which $\mathcal{D}$ is as shown in Figure 1. Let $A_1 = \begin{pmatrix} 0.2 & 0.4 & 0.8 \\ 0.3 & 0.6 & 0.9 \\ 0.5 & 1.5 & 1.5 \end{pmatrix}$, $A_2 = \begin{pmatrix} -1 & 0.1 & 0.2 \\ 0.3 & -1 & 0.4 \\ 0.5 & 0.6 & -1 \end{pmatrix}$, $A_3 = \begin{pmatrix} -0.1 & -0.2 & 0.3 \\ -0.1 & -0.4 & 0.6 \\ 0.8 & 0.7 & -0.6 \end{pmatrix}$.

Notice that the matrices $A_1, A_2$ and $A_3$ are full-rank. The following steps are carried out: 1) Set $Q = T = \{\varepsilon\}$.
2) Apply Algorithm 1 to all $w \in \{\varepsilon, fault, ideal\}$. It is observed that $(Q,T)$ is $\mathcal{D}$-separable but not $\mathcal{D}$-closed. Indeed, $\mathit{Output}_{\mathcal{D}}(\varepsilon \cdot ideal) \neq \mathit{Output}_{\mathcal{D}}(\varepsilon)$. Update $Q = \{\varepsilon, ideal\}$.
3) Apply Algorithm 1 to all $w \in \{ideal\cdot fault, ideal\cdot ideal\}$. It is observed that $(Q,T)$ is $\mathcal{D}$-separable and $\mathcal{D}$-closed. Construct the hypothesis FA $\mathcal{D}'$ shown in Figure 2. Checking for correctness of $\mathcal{D}'$ with the counter-example generator, yields a counter-example $w = ideal \cdot ideal \cdot ideal$. Update $Q = \{\varepsilon, ideal, ideal \cdot ideal\}$ and $T = \{\varepsilon, ideal\}$.
4) Apply Algorithm 1 to all $w \in \{ideal \cdot fault, ideal \cdot ideal, fault \cdot ideal, ideal \cdot fault \cdot ideal, ideal \cdot ideal \cdot ideal, ideal \cdot ideal \cdot fault, ideal \cdot ideal \cdot fault \cdot ideal, ideal \cdot ideal \cdot ideal \cdot fault\}$. It is observed that $(Q,T)$ is $\mathcal{D}$-separable but not $\mathcal{D}$-closed. Indeed, $\mathit{Output}_{\mathcal{D}}(ideal \cdot ideal \cdot ideal) \neq \mathit{Output}_{\mathcal{D}}(\varepsilon)$, $\mathit{Output}_{\mathcal{D}}(ideal\cdot ideal\cdot ideal) \neq \mathit{Output}_{\mathcal{D}}(ideal)$, $\mathit{Output}_{\mathcal{D}}(ideal \cdot ideal \cdot ideal) \neq \mathit{Output}_{\mathcal{D}}(\varepsilon \cdot \varepsilon)$. Update $Q = \{\varepsilon, ideal, ideal \cdot ideal, ideal \cdot ideal \cdot ideal\}$.
5) Apply Algorithm 1 to all $w \in \{ideal \cdot ideal \cdot ideal \cdot fault, ideal \cdot ideal \cdot ideal \cdot fault \cdot ideal, ideal \cdot ideal \cdot ideal \cdot ideal, ideal \cdot ideal \cdot ideal \cdot ideal \cdot ideal\}$. It is observed that $(Q,T)$ is $\mathcal{D}$-separable and $\mathcal{D}$-closed. Construct the hypothesis FA $\mathcal{D}'$ shown in Figure 3. Checking for correctness of $\mathcal{D}'$ with the counter-example generator does not yield a counterexample. We conclude that $\mathcal{D}'$ is a FA that accepts the language, $L_{\mathcal{D}}$.

*Remark 5:* The FA considered above resembles the automata used to implement $L^*$-algorithm in [10, §2] without final nodes and with node labels. We note that the total number of membership queries and equivalence queries required for the learning task in [10, Section 2] matches the total number of calls to Algorithm 1 and the check for correctness of hypothesis FA in our setting.

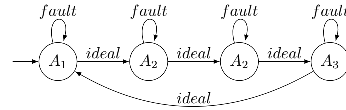


Fig. 1. FA

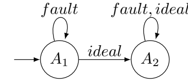


Fig. 2. Hypothesis automaton $\mathcal{D}'$ in Step 3.

## V. Conclusion

In this paper, we presented a learning algorithm for the identification of event-driven switched linear systems. Our future research directions include the design of active learning techniques for large-scale switched systems whose subsystems dynamics are not restricted to be linear structures and/or the available state-trajectories are noisy.

## References

[1] D. Angluin, *Learning regular sets from queries and counterexamples*, Inform. and Comput., 75 (1987), pp. 87–106.
[2] D. Angluin, B. Becerra-Bonache, A. H. Dediu, and L. Reyzin, *Learning finite automata using label queries*, Proceedings of the 20th International Conference on Algorithmic Learning Theory, (2009), pp. 171–185.
[3] M. D. Comparetti, E. Beretta, M. Kunze, E. D. Momi, J. Raczkowsky, and G. Ferrigno, *Event-based device-behavior switching in surgical human-robot interaction*, IEEE International Conference on Robotics and Automation (ICRA), (2014), pp. 1877–1882.
[4] Z. Du, L. Balzano, and N. Ozay, *A robust algorithm for online switched system identification*, IFAC Symposium on System Identification (SYSID), (2018), pp. 293–298.
[5] A. Garulli, S. Paoletti, and A. Vicino, *A survey on switched and piecewise affine system identification*, IFAC Symposium on System Identification, (2012), pp. 344–355.
[6] A. Kundu and P. Prabhakar, *Learning event-driven switched linear systems*, 2020. arxiv: 2009.12831.
[7] F. Lauer, *On the complexity of switching linear regression*, Automatica J. IFAC, 74 (2016), pp. 80–83.
[8] D. Liberzon, *Switching in Systems and Control*, Systems & Control: Foundations & Applications, Birkhäuser Boston Inc., Boston, MA, 2003.
[9] S. Paoletti, A. L. Juloski, G. Ferrari-Trecate, and R. Vidal, *Identification of hybrid systems: A tutorial*, European Journal of Control, 13 (2007), pp. 242–260.
[10] J. Worrell, *Exactly learning regular languages using membership and equivalence queries*, 2017. Available at https://www.cs.ox.ac.uk/people/james.worrell/DFA-learning.pdf.

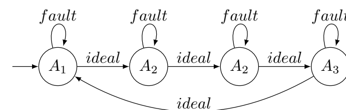


Fig. 3. Hypothesis automaton $\mathcal{D}'$ in Step 5.



Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on May 12,2023 at 05:18:02 UTC from IEEE Xplore. Restrictions apply.