Fundamental results for learning deterministic extended finite state machines from queries

Florentin Ipate[a], Marian Gheorghe[b] and Raluca Lefticaru[b]

[a] *Department of Computer Science, Faculty of Mathematics and Computer Science and ICUB, University of Bucharest Str. Academiei 14, sector 1, 010014, Bucharest, Romania; Email: florentin.ipate@ifsoft.ro*

[b] *Department of Computer Science, Faculty of Engineering and Informatics, University of Bradford, Bradford BD7 1DP, United Kingdom; Email: {m.gheorghe, r.lefticaru}@bradford.ac.uk*

Abstract

Regular language inference, initiated by Angluin, has many developments, including applications in software engineering and testing. However, the capability of finite automata to model the system data is quite limited and, in many cases, extended finite state machine formalisms, that combine the system control with data structures, are used instead. The application of Angluin-style inference algorithms to extended state machines would involve constructing a minimal deterministic extended finite state machine consistent with a deterministic 3-valued deterministic finite automaton. In addition to the usual, accepting and rejecting, states of finite automaton, a 3-valued deterministic finite automaton may have "don't care" states; the sequences of inputs that reach such states may be considered as accepted or rejected, as is convenient. The aforementioned construction reduces to finding a minimal deterministic finite automaton consistent with a 3-valued deterministic finite automaton, that preserves the deterministic nature of the extended model that also handles the data structure associated with it. This paper investigates fundamental properties of extended finite state machines in relation to Angluin's language inference problem and provides an inference algorithm for such models.

Keywords: 3DFA, finite automata, learning from queries, extended finite state machines, X-machines

1. Introduction

Finite automata are widely investigated formalisms [1] with well-known applications in programming languages specification and implementation [2], system design, with a plethora of methods and techniques utilised in their formal analysis, verification [3] and testing [4]. In many circumstances, such models are not produced or maintained during system development and the only way of generating them is through inference, by examining the external behavior of the implementation.

Regular language inference was initiated by Angluin [5], who also introduces a learning algorithm, called L^*. A number of variants of the L^* algorithm have been introduced and investigated: domain-specific optimizations [6], a modified algorithm whose complexity depends on the representation of the deterministic finite automaton rather than on the size of the alphabet [7] and, more recently, a learning algorithm for cover automata [8]. Existing publications outline the importance of this research in software engineering applications [9], including industrial automation systems and testing [10].

While finite automata can successfully model the control aspects of a system, their capability to model the system data is quite limited. On the other hand, *extended finite state machine* (*EFSM*, for short) formalisms, that combine the system control with data structures, exist and can be used to alleviate this limitation. This model is successfully used in model based testing of interactive systems [11]. A survey on the use of EFSM model for test case generation can be found in [12]. In order to apply the automata based techniques to a system modelled by an EFSM, a finite automaton has to be abstracted from the actual EFSM, which may prove, technically, a problematic transformation. In particular, as shown in section 4, the application of Angluin-style inference algorithms to EFSM would involve constructing a minimal deterministic EFSM consistent with a 3-valued deterministic finite automaton (*3DFA*, for short). In addition to the usual, accepting and rejecting, states of finite automaton, a 3DFA may have "don't care" states; the sequences of inputs that reach such states may be considered as accepted or rejected, as is convenient. The aforementioned construction reduces to finding a minimal DFA consistent with a 3DFA, that preserves the deterministic nature of the extended model. The problem is non-trivial since the extended model contains, in addition to states and transition between states, a memory structure and rules for updating the memory associated with transitions. This is addressed in this paper.

The type of EFSM used here is the stream X-machine (*SXM*, for short). The SXM model has been investigated for both theoretical aspects as well as its potential to be applied in model based testing [13]. SXM also has benefitted from a set of robust test generation methods [14, 15, 16, 17, 18, 19, 20]; model based testing is an important means used to improve dependability of critical systems [21]. SXMs are used as underlying formal models in agent-based systems and the simulation framework called FLAME, that has many important applications, ranging from biology [22, 23] to macroeconomy [24].

The SXM model is also linked with membrane computing. Several classes of P systems [25] have been studied in connection with the SXM model [26] showing the potential of transferring methods from one domain to the other [27, 28].

The remainder of the paper is structured as follows. Sections 2 and 3 present background information. Section 4 introduces the stream X-machine model and clarifies the motivation of the paper (the application of Angluin-style inference algorithms to stream X-machines). The following three sections, that constitute the novel contribution of the paper, provide results and algorithms for constructing a minimal deterministic stream X-machine consistent with a deterministic 3DFA. Conclusions are drawn and future work is outlined in section 8.

2. Preliminaries

Basic notations and concepts used in the paper are introduced below. Given a finite set A, called *alphabet,* A^* is the set of all finite sequences of symbols over A with ϵ being the empty sequence. For a, b two sequences from A^*, their concatenation is denoted by ab. If a sequence $a \in A^*$, is such that $a = bc$, with $b, c \in A^*$, then b is called a *prefix* of a and c is a *suffix* of a. For $U \subseteq A^*$, the complement of U, denoted $\overline{U}$, is given by $\overline{U} = A^* \setminus U$.

Deterministic finite automata and related concepts and results to be later investigated in the paper are now briefly introduced.

$M = (A, Q, h, q_0, F)$ is called *deterministic finite automaton* (*DFA*, for short), where A, Q, F, with $F \subseteq Q$, are finite sets called *input alphabet*, *set of states*, and *set of final states*, respectively; h is the *next-state* function, $h : Q \times A \longrightarrow Q$, and $q_0 \in Q$ is the *initial state.*

The function h is usually extended to sequences over A, $h' : Q \times A^* \longrightarrow Q$, where $h'(q, \epsilon) = q$, $q \in Q$, and $h'(q, sa) = h(h'(q, s), a)$, with q a state in Q, s a sequence over A and a a symbol in A. For simplicity, h will be used in both cases.

For $q \in Q$, we define the set $L_M^q = \{s \in A^* \mid h(q, s) \in F\}$. When $q = q_0$, $L_M^{q_0}$ is called the *language accepted by* M, simply denoted by L_M. A state q from Q is called *reachable* if there is a sequence s from A^* such that $h(q_0, s) = q$. A DFA M is called *reachable* if all its states are reachable. A DFA M is called *reduced* if any two distinct states q_1 and q_2 have the property $L_M^{q_1} \neq L_M^{q_2}$. A DFA M is called *minimal* if any DFA accepting L_M has the number of states greater than or equal to those of M.

3. Background - learning regular languages from queries

In the set-up of Angluin's L^* algorithm, a *learner* will infer an unknown regular language $U \subseteq A^*$ over a known alphabet A by asking questions to a *teacher* and an *oracle.* There are two kinds of questions:

- *Membership queries* - the learner checks with the teacher whether certain input sequences belong to U and the results of the queries are kept in an *observation table.* Periodically, based on the observation table, a DFA is built.
- *Equivalence queries* - the learner asks the oracle whether the constructed DFA accepts U. The oracle answers "yes" when M is the correct DFA. Otherwise, it provides a sequence t (called counterexample), which is in one of U or L_M, but not in both. The observation table is then modified based on the counterexample t.

Eventually, a minimal DFA for U is produced.

The set of input sequences in the observation table is given by $(S \cup SA)W$, where S is a non-empty, finite, prefix-closed set of sequences, A is the above mentioned alphabet, and W is a non-empty, finite, suffix-closed set of sequences. The *observation table* is represented as a mapping $O : (S \cup SA)W \longrightarrow \{0, 1\}$ such that $O(u) = 1$ when $u \in U$ and $O(u) = 0$ for $u \notin U$.

The table can be also represented as a two-dimensional array having rows labelled by elements of $S \cup SA$ and columns labelled by elements of W. In this case, the value of the array corresponding to row labelled by elements of $s \in S \cup SA$ and column labelled by elements of $w \in W$ is $O(sw)$. The row with values 0 or 1, and labelled s, $s \in S \cup SA$, is denoted by $row(s)$. Initially in the observation table, we have $S = W = \{\epsilon\}$.

The algorithm uses the concepts of consistent and closed observation table. An observation table is *consistent* when for any $s_1, s_2 \in S$ such that $row(s_1) = row(s_2)$, then $row(s_1a) = row(s_2a)$, for any symbol a in A. An observation table is *closed* when for any sequence s in SA, there exists a sequence t in S such that $row(s) = row(t)$.

For a consistent and closed observation table, the algorithm constructs the corresponding DFA, $M(S, W, O) = (A, Q, h, q_0, F)$, where $Q = \{row(s) \mid s \in S\}$; $h(row(s), a) = row(sa)$, $s \in S$, $a \in A$; $q_0 = row(\epsilon)$; $F = \{row(s) \mid s \in S, O(s) = 1\}$.

The algorithm extends the observation table whenever one of the following three situations occurs: the table is not consistent, the table is not closed or the table is both consistent and closed but the resulting automaton $M(S, W, O)$ does not accept U (in which case a counterexample is produced). Each time the observation table is extended as a result of an incorrect consistency or closedness check, the number of distinct rows increases. The reader is referred to [5] for the full description of the L^* algorithm and further details.

A more general case, in which not all the answers provided by the teacher are relevant for the automaton to be learned, has also been studied [29, 30, 31, 32]. In addition to "yes" and "no" the teacher can also respond by "don't care" to membership queries. More formally, let $U_1, U_2 \subseteq A^*$ be two disjoint regular languages. A deterministic finite automaton M is said to *separate* U_1 and U_2 if the language L_M accepted by M contains U_1 and is disjoint from U_2, i.e. $U_1 \subseteq L_M$ and $L_M \cap U_2 = \emptyset$ ($U_1 \subseteq L_M \subseteq \overline{U_2}$). M is called a *minimal separating DFA* of U_1 and U_2 if M has the minimum number of states among all DFAs separating U_1 and U_2. The most computationally efficient approach for this scenario is the L^{Sep} algorithm [29], that uses the concept of 3-valued deterministic finite automaton.

Definition 3.1. *A* 3-valued deterministic finite automaton *(or* 3DFA*) is a tuple $C = (A, Q, h, q_0, Acc, Rej, Dont)$, where A, Q, q_0, h are the components of a deterministic finite automaton and the state set Q is partitioned into three sets: Acc, Rej and $Dont$, denoting accepting states, rejecting states and don't care states, respectively.*

A sequence $s \in A^*$ is accepted by C if $h(q_0, s) \in Acc$, is rejected by C if $h(q_0, s) \in Rej$ and is a don't care sequence if $h(q_0, s) \in Dont$. Let us consider the following DFAs, $C^+ = (A, Q, h, q_0, Acc \cup Dont)$, with the don't care states becoming accepting states, and $C^- = (A, Q, h, q_0, Acc)$, with the don't care states being rejecting states. L_{C^-} corresponds to the set of accepted sequences in C and $\overline{L_{C^+}}$ represents the set of rejected sequences in C.

Definition 3.2. *A DFA M with the same input alphabet as the 3DFA C is* consistent *with C if M accepts all the sequences accepted by C and rejects all the sequences rejected by M, i.e. $L_{C^-} \subseteq L_M \subseteq L_{C^+}$.*

A *minimal DFA consistent* with a 3DFA C is a DFA consistent with C and having a minimum number of states.

Given two disjoint regular languages, U_1 and U_2, the L^{Sep} algorithm finds a minimal separating DFA of U_1 and U_2 by first inferring a 3DFA C from the samples collected from U_1 and U_2. C is constructed by accepting all sequences in U_1 and rejecting all sequences in U_2, whereas the remaining sequences will take C to don't care states.

Similarly to the L^* algorithm, in the L^{Sep} algorithm a learner asks two types of questions to a teacher and an oracle:

- *Membership queries*, answered to by the teacher. Unlike in the case of the L^* algorithm, there are three possible answers to a membership query: "true", if the input sequence is in U_1, "false", if the input sequence is in U_2 and "don't care", otherwise.

- *Containment queries*, answered to by the oracle. There are four types of containment queries: (1) $U_1 \subseteq L_M$, (2) $L_M \subseteq U_1$, (3) $\overline{U_2} \subseteq L_M$, (4) $L_M \subseteq \overline{U_2}$, where M is the conjecture DFA. The oracle will answer "yes" if M satisfies the containment query, or else supply a counterexample.

The L^{Sep} algorithm involves the following steps [29]:

- *Candidate generation.* In this step a 3DFA candidate is generated from membership queries by extending the table in L^* to allow entries with don't care results. In this case, the function O will take three values: $O(u) = 1$ if $u \in U_1$, $O(u) = 0$ if $u \in U_2$ and $O(u) = -1$ otherwise (-1 is used for don't care). Similarly to the L^* algorithm, with a closed and consistent table, the algorithm will produce a candidate 3DFA, $C(S, W, O)$, which will be checked for completeness (second step of the algorithm) and soundness (fourth step of the algorithm). If the completeness or soundness checks fail, the algorithm will extend the observation table by using the received counterexample and will eventually produce a new candidate 3DFA.

- *Completeness checking.* In this step, the algorithm checks if $L_{C(S,W,O)^-} \subseteq U_1$ and $\overline{U_2} \subseteq L_{C(S,W,O)^+}$. If either of these queries fails, a counterexample is produced and sent to candidate generation to refine the conjecture

3DFA. Several iteration steps involving candidate generation and completeness checks may be needed before a candidate 3DFA that satisfies the two conditions above is produced. As the L^{Sep} algorithm reduces the problem of finding a minimal separating DFA of U_1 and U_2 to finding a minimal DFA consistent with $C(S, W, O)$, this step ensures that all separating DFAs of U_1 and U_2 are considered.

- *Finding a minimal consistent DFA.* The next step of the algorithm consists of finding a minimal DFA $M(S, W, O)$ consistent with $C(S, W, O)$. The L^{Sep} algorithm reduces this problem to the minimization problem of incompletely specified finite state machines and invokes the algorithm in [33] for this purpose.

- *Soundness checking.* The fourth step of the algorithm checks whether $M(S, W, O)$ is a separating DFA of U_1 and U_2 by using the containment queries $U_1 \subseteq L_{M(S,W,O)}$ and $L_{M(S,W,O)} \subseteq \overline{U_2}$. If both checks succeed then $M(S, W, O)$ is the minimal separating DFA of U_1 and U_2; otherwise, a counterexample is produced and sent to the candidate generator to refine the 3DFA and start a new iteration.

The algorithm is polynomial in the number of membership queries and the length of the longest counterexample. However, finding a minimal DFA consistent with a 3DFA is computationally expensive and so a heuristic that finds a "reduced" (but not necessarily minimal) DFA consistent with a 3DFA is also proposed. For further details the reader is referred to [29].

4. Motivation - learning stream X-machines from queries

A stream X-machine consists of a finite automaton, representing the control component, and a data store, called *memory.* Like a finite state machine, a stream X-machine processes input sequences producing output sequences. However, the transition labels do not indicate abstract symbols, like in a finite automaton, but *processing functions.* Each processing function, a partially or completely defined function representing an elementary operation, reads an input and, depending on the current memory value, produces an output and updates the memory value.

Definition 4.1. *A* stream X-Machine *(abbreviated as* SXM*) is a tuple* $Z = (\Sigma, \Gamma, Q, Mem, \Phi, h, q_0, m_0)$, *where:*

- Σ *is a finite set called* input alphabet*;*
- Γ *is a finite set called* output alphabet*;*
- Q *is the finite set of* states*;*
- Mem *is a set called* memory*;*

- Φ *is a finite set of distinct* processing functions*; a processing function is a non-empty (possibly partial) function of type* $Mem \times \Sigma \longrightarrow \Gamma \times Mem$*;*
- h *is the (possibly partial)* next-state function*,* $h : Q \times \Phi \longrightarrow Q$*;*
- $q_0 \in Q$ *is the initial state;*
- $m_0 \in Mem$ *is the initial memory value.*

Intuitively, a SXM Z can be regarded as a deterministic finite automaton with transition labels indicating functions from the set Φ. The automaton $M_Z = (\Phi, Q, h, q_0, Q)$, where Φ is the alphabet of function labels, is called the *associated deterministic finite automaton* (abbreviated as *associated DFA*) of Z.

The set Φ is called the *type* of Z. When a SXM Z is used as a model of a software system, each processing function of Φ specifies components used in the software system, such as basic operations, more complex components or even sub-systems. The memory normally represents the variable values used by the computer program. Often, Mem is defined as a set of tuples, where each component indicates either a global variable or a parameter that may be passed between the elements of Φ [13].

Note 4.1. *Mem may be potentially infinite, but in practice it is always finite and this is the assumption we make in this paper, i.e. Mem is finite.*

Note that, as defined above, M_Z does not have explicit rejecting states, but the next-state function h may be a *partial* function, so the rejected sequences of processing functions are those that cannot be traced out from the initial state q_0. For consistency with the definition of a DFA from section 2, we introduce a rejecting state $rej \notin Q$ that "collects" all the undefined transitions and so the associated DFA M_Z will be the tuple $M_Z = (\Phi, Q \cup \{rej\}, h, q_0, Q)$, with h completely-defined. As with any automaton, the function h may be extended to sequences from Φ^* and one can define $L^q_{M_Z} = \{p \in \Phi^* \mid h(q, p) \in Q\}$. Also, similarly to finite automata, when $q = q_0$, this will be called the *language accepted* by M_Z and denoted by L_{M_Z}. This includes all the paths from q_0.

A sequence p of processing functions induces a (partial) function $\|p\|$ that describes the relationship between a (memory value, input sequence) pair and an (output sequence, memory value) pair produced by the application, in turn, of the processing functions in the sequence p. More formally, given $p \in \Phi^*$, $\|p\| : Mem \times \Sigma^* \longrightarrow \Gamma^* \times Mem$ is defined by: $\|\epsilon\|(m, \epsilon) = (\epsilon, m)$, $m \in Mem$; for $p \in \Phi^*$ and $\phi \in \Phi$, $\|p\phi\|(m, s\sigma) = (g\gamma, m')$, for $m, m' \in Mem, s \in \Sigma^*, g \in \Gamma^*, \sigma \in \Sigma, \gamma \in \Gamma$ such that there exists $m'' \in Mem$ with $\|p\|(m, s) = (g, m'')$ and $\phi(m'', \sigma) = (\gamma, m')$.

A machine computation takes all accepted sequences of processing functions associated with transitions starting from the initial state and then applies it to the initial memory value. This gives rise to the *relation (function) computed* by Z, denoted f_Z, that links input sequences processed by the sequences of processing functions to the output sequences produced. More formally, f_Z :

$\Sigma^* \longleftrightarrow \Gamma^*$ contains all pairs (s, g), $s \in \Sigma^*$, $g \in \Gamma^*$, with the property that there exist $p \in \Phi^*$ and $m \in Mem$ such that $h(q_0, p) \in Q$ and $\|p\|(m_0, s) = (g, m)$.

A *deterministic SXM* (abbreviated *DSXM*) is a SXM with at most one possible accepted transition for any triplet (state, memory, input). More formally, a deterministic SXM Z has the property that for every $\phi_1, \phi_2 \in \Phi$, and $q \in Q$, if $h(q, \phi_1) \in Q$ and $h(q, \phi_2) \in Q$ then either $\phi_1 = \phi_2$ or $dom\ \phi_1 \cap dom\ \phi_2 = \emptyset$. When Z is deterministic, its associated DFA M_Z is called Φ-*deterministic*. A DSXM will compute a (partial) function f_Z. In this paper we only consider deterministic stream X-machines.

Naturally, not all sequences of processing functions of a DSXM can be associated to input sequences. For a memory value m, a sequence of processing functions triggered from m by an input sequence is called *realizable* in m or simply realizable when $m = m_0$ [18]. The set of realizable sequences in m is denoted by $R_\Phi(m)$ and when $m = m_0$ the notation is R_Φ. More formally, the set $R_\Phi(m) \subseteq \Phi^*$ consists of all sequences $p = \phi_1 \dots \phi_n \in \Phi^*$, $n \geq 0$, for which there exists $s \in \Sigma^*$ such that $(m, s) \in dom\ \|p\|$.

Note 4.2. *The realizable sequences of functions $\phi \in \Phi$ that appear in R_Φ can be described by a finite automaton having as states memory values from Mem, as this is finite (see Note 4.1), and transitions labelled by functions ϕ. Indeed, whenever ϕ is applied for an input $\sigma \in \Sigma$ and a memory $m \in Mem$, yielding and output $\gamma \in \Gamma$ and a new memory value $m' \in Mem$, i.e. $\phi(m, \sigma) = (\gamma, m')$, one can define a transition from m to m' labelled ϕ. Hence, R_Φ is a regular language.*

The stream X-machine model has been successfully used as a basis for test generation and a number of such methods exist. These methods identify certain design constraints for the specification, referred to as *design for test conditions*, that facilitate the testing process. Naturally, different DSXM based testing methods may ask for different design for test conditions, of different strength, but all methods require, quite naturally, the tester to be able to determine the sequence of processing functions applied in the implementation under test to a given input sequence by examining the output sequence produced. This requirement, imposed on the DSXM specification, is called output-distinguishability.

Definition 4.2. Φ *is said to be* output-distinguishable *if for all $\phi_1, \phi_2 \in \Phi$, whenever there exist $m, m_1, m_2 \in Mem, \sigma \in \Sigma, \gamma \in \Gamma$ such that $\phi_1(m, \sigma) = (\gamma, m_1)$ and $\phi_2(m, \sigma) = (\gamma, m_2)$, we have $\phi_1 = \phi_2$.*

Let us briefly examine how the previously presented methods for learning regular languages can be applied to stream X-machines. More specifically, suppose a learner is trying to learn the minimal DSXM of an unknown DSXM Z with known type output-distinguishable Φ and initial memory value m_0 by asking queries to a teacher and an oracle.

Analogously to the L^* algorithm, two kinds of queries can be used:

- *Value queries* - the learner is asking the teacher the value of the function f_Z for a given input sequence s. On the basis of the results produced by the teacher, periodically, a DSXM Z' will be constructed.

- *Equivalence queries* - the learner is asking the oracle whether the constructed DSXM Z' is correct, i.e. it computes f_Z. The answer of the oracle is "yes" when the constructed DSXM Z' is correct, otherwise a counterexample t, such that $f_Z(t) \neq f_{Z'}(t)$, is supplied.

Analogously to the learning algorithms for DFA presented above, the DSXM learning algorithm relies on an observation table, associated with a function O mapping finite sequences of processing functions to some distinct values; it will transpire that at most three values are needed. The table will be given by a two-dimensional array with rows corresponding to values from $S \cup S\Phi$ and columns given by elements of the set W, where S, a subset of Φ^*, is a non-empty, finite, prefix-closed set of sequences of processing functions and W, a subset of Φ^*, is a non-empty, finite, suffix-closed set of sequences of processing functions. Initially in the observation table we have $S = W = \{\epsilon\}$.

Consider a sequence of processing functions, $p \in (S \cup S\Phi)W$. In order to establish the value of $O(p)$, the algorithm will decide if $p \in R_\Phi$. The following cases can be then distinguished:

- $p \notin R_\Phi$. In this case it does not matter if $p \in L_{M_Z}$ since p does not contribute to the computed function f_Z. Consequently, we assign $O(p)$ the don't care value -1, i.e. $O(p) = -1$.

- $p \in R_\Phi$. In this case the algorithm will construct input sequence s such that $(m_0, s) \in dom\ \|p\|$ and will ask the teacher for the value $g = f_Z(s)$. Let $\|p\|(m_0, s) = (g', m)$, $g' \in \Gamma$, $m \in Mem$. Since Φ is output-distinguishable, $p \in L_{M_Z}$ if and only if $g = g'$ [13]. Then $O(p) = 1$ if $g = g'$ (as it is known that $p \in L_{M_Z}$) and $O(p) = 0$ otherwise (as it is known that $p \notin L_{M_Z}$).

Now, suppose that the algorithm has constructed a conjecture DSXM Z'. Then the oracle will be asked if Z is correct and, otherwise, a counterexample t, such that $f_Z(t) \neq f_{Z'}(t)$, will be produced. Let $f_Z(t) = g$ and $f_{Z'}(t) = g'$. Let $p \in L_{M_{Z'}}$ be such that $\|p\|(m_0, t) = (g', m)$ for some $m \in Mem$. As $g \neq g'$ and Φ is output-distinguishable, it follows that $p \notin L_{M_Z}$ Hence $p \in L_{M_{Z'}} \setminus L_{M_Z}$.

Note 4.3. *From the above observations, it looks as though the DSXM learning algorithm can be reduced to the L^{Sep} algorithm for learning a minimal separating DFA of U_1 and U_2, where $U_1 = L_{M_Z} \cap R_\Phi$ and $U_2 = \overline{L_{M_Z}} \cap R_\Phi$ (according to Note 4.2, R_Φ is regular, hence the above language is regular).*

Note 4.4. *However, the application of the L^{Sep} algorithm to DSXMs is not straightforward since finding a minimal DFA consistent with a 3DFA (the 3rd step of the algorithm) may yield a* non-deterministic SXM*, as shown next.*

As the associated DFA of a DSXM has one rejecting state, the 3DFAs that result from the application of the (extended) Angluin algorithm will also have one rejecting state rej.

Note that, if a sequence $p \in \Phi^*$ of processing functions is not realizable, then $pp' \in \Phi^*$ is not realizable either for every $p' \in \Phi^*$, so any transition from a *Dont* state will also go to a *Dont* state. Thus, one don't state will suffice, so we can consider that $Dont = \{dont\}$ and $h(dont, \phi) = dont$ for all $\phi \in \Phi$.

Definition 4.3. *Let $C = (\Phi, Q \cup \{rej\}, h, q_0, Acc, \{rej\}, \{dont\})$, $Q = Acc \cup \{dont\}$, be a 3DFA over the set of labels of the processing functions Φ. C is called* Φ-deterministic *if for every $\phi_1, \phi_2 \in \Phi$, and $q \in Q$ such that $h(q, \phi_1) \in Acc$ and $h(q, \phi_2) \in Acc$ then either $\phi_1 = \phi_2$ or dom $\phi_1 \cap$ dom $\phi_2 = \emptyset$.*

One can observe that a Φ-deterministic 3DFA is also deterministic.

Example 4.1. *A Φ-deterministic 3DFA, C_1, and a minimal DFA, M_1, consistent with C_1, but not Φ-deterministic are built.*

Let $C_1 = (\Phi, Q \cup \{rej\}, h, q_0, Acc, \{rej\}, \{dont\})$ be as follows. Let $\Sigma = \{a, b\}$, $\Gamma = \{x, y\}$, $Mem = \{0, 1, 2, 3\}$. Let $\Phi = \{\phi_1, \phi_2, \phi_3, \phi_4\}$ with $\phi_1, \phi_2, \phi_3, \phi_4$ defined by:

$\phi_1(0, a) = (x, 1)$, $\phi_1(2, a) = (x, 3)$;
$\phi_2(1, a) = (y, 2)$, $\phi_2(3, a) = (y, 3)$;
$\phi_3(0, b) = (x, 1)$, $\phi_3(1, b) = (x, 1)$, $\phi_3(2, b) = (x, 2)$;
$\phi_4(0, b) = (y, 1)$, $\phi_4(3, b) = (y, 3)$.

Let $Acc = \{q_0, q_1, q_2, q_3\}$ and h as defined in Figure 1. Also, for any $\phi_i \in \Phi$, $1 \leq i \leq 4$, the next-state function, h, is defined in dont by $h(dont, \phi_i) = dont$, $1 \leq i \leq 4$. Let $M_1 = (\Phi, Q' \cup \{rej'\}, h', q_0', Q')$ with $Q' = \{q_0', q_1'\}$ and h' defined as in Figure 2. M_1 is consistent with C_1 and, furthermore, M_1 is a minimal DFA. It can be observed that dom $\phi_i \cap$ dom $\phi_j = \emptyset$, $1 \leq i < j \leq 4$, $(i, j) \neq (3, 4)$ and dom $\phi_3 \cap$ dom $\phi_4 \neq \emptyset$. Since for every i, $0 \leq i \leq 3$, $h(q_i, \phi_3) \in Acc$ and $h(q_i, \phi_4) \in Acc$ do not hold simultaneously, C_1 is a Φ-deterministic 3DFA. On the other hand, since $h'(q_1', \phi_3) = q_1'$ and $h'(q_1', \phi_4) = q_1'$, M_1 is not a Φ-deterministic DFA.

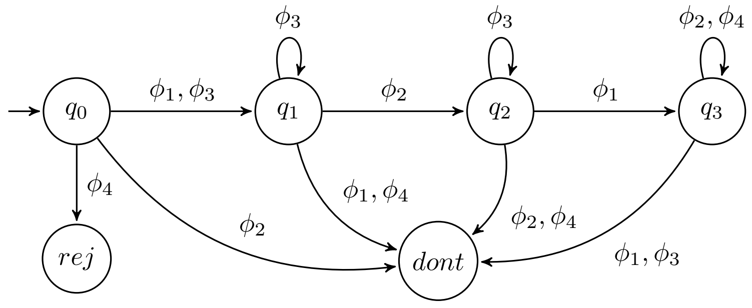

Figure 1: State transition diagram of C

The remainder of the paper investigates the construction of a minimal Φ-deterministic DFA consistent with a Φ-deterministic 3DFA. The problem to

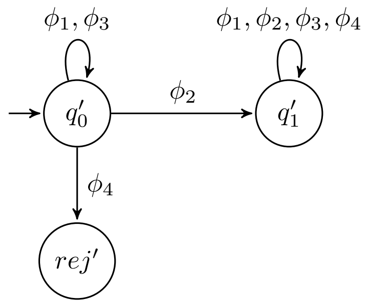

Figure 2: State transition diagram of M_1

.

be addressed is as follows. Let Φ be a set of processing functions. In this section we will use DFAs and 3DFAs over the set of labels of Φ. Let $C = (\Phi, Q \cup \{rej\}, h, q_0, Acc, \{rej\}, \{dont\})$, $Q = Acc \cup \{dont\}$, be a Φ-deterministic 3DFA over the set of labels of Φ. We assume, without loss of generality, that every state of C is reachable by some sequence from the initial state (otherwise it can be removed without changing the languages accepted by C). We need to construct a minimal Φ-deterministic DFA M that is consistent with C, that is $L_{C^-} \subseteq L_M \subseteq L_{C^+}$ (Definition 3.2). (A minimal Φ-deterministic DFA consistent with C is a DFA that has minimum states among all Φ-deterministic DFAs consistent with C).

5. Closed and domain-consistent decompositions

In our construction, we will use closed and domain-consistent decompositions of the state space, as defined below.

Definition 5.1. *Let $C = (\Phi, Q \cup \{rej\}, h, q_0, Acc, \{rej\}, \{dont\})$, $Q = Acc \cup \{dont\}$, be a Φ-deterministic 3DFA as above and $M = (\Phi, Q' \cup \{rej'\}, h', q'_0, Q')$ a Φ-deterministic DFA. Given state $q' \in Q' \cup \{rej'\}$ of M and state $q \in Q \cup \{rej\}$ of C, we say that q'* covers *q if, for every $p \in \Phi^*$, (1) $h(q, p) = rej$ implies $h'(q', p) = rej'$; (2) $h(q, p) \in Acc$ implies $h'(q', p) \in Q'$. We say that M covers C if q'_0* covers *q_0.*

From this definition it follows that *dont* is covered by any state of M.

Let C be as in the first running example. Let $M = (\Phi, Q' \cup \{rej'\}, h', q'_0, Q')$ with $Q' = \{q'_0, q'_1, q'_2\}$ and h' defined as in Figure 3. Then q'_0 covers q_0, q_1 and *dont*, q'_1 covers q_1, q_2 and *dont*, q'_2 covers q_3 and *dont*, rej' covers rej and *dont*. As q'_0 covers q_0, M covers C.

Note 5.1. *Given a subset of states $R \subseteq Q \cup \{rej\}$ and $\phi \in \Phi$, we denote $h(R, \phi) = \{h(q, \phi) \mid q \in R\}$.*

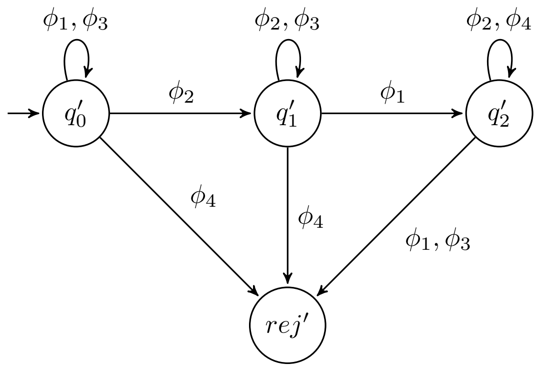

Figure 3: State transition diagram of M

.

The following definitions will refer to a $\Phi-$deterministic 3DFA as defined above.

Definition 5.2. *States q_1 and q_2, $q_1, q_2 \in Q \cup \{rej\}$, are said to be* domain-compatible *if, for every $\phi_1, \phi_2 \in \Phi$, the following holds: if $h(q_1, \phi_1) \in Acc$ and $h(q_2, \phi_2) \in Acc$ then either $\phi_1 = \phi_2$ or dom $\phi_1 \cap$ dom $\phi_2 = \emptyset$.*

Definition 5.3. *Let $Q_1, Q_2, \dots Q_n$, $Q_i \subseteq Q \cup \{rej\}$, $1 \leq i \leq n$, be non-empty subsets of states of C (not necessarily disjoint). Then $D = \{Q_1, Q_2, \dots Q_n\}$ is called a* decomposition *of $Q \cup \{rej\}$ if (1) for every i, $1 \leq i \leq n$, either $Q_i \subseteq Q$ or $Q_i \subseteq \{dont, rej\}$ and (2) $Q_1 \cup Q_2 \cup \dots \cup Q_n = Q \cup \{rej\}$. Q_i is called an accepting subset if $Q_i \subseteq Q$ and Q_i is called a rejecting subset if $Q_i \subseteq \{dont, rej\}$.*

Definition 5.4. *A decomposition $D = \{Q_1, Q_2, \dots Q_n\}$ of $Q \cup \{rej\}$ is said to be* closed *if for every i, $1 \leq i \leq n$, and every $\phi \in \Phi$, there exists j, $1 \leq j \leq n$, such that $h(Q_i, \phi) \subseteq Q_j$.*

Definition 5.5. *A decomposition $D = \{Q_1, Q_2, \dots Q_n\}$ of $Q \cup \{rej\}$ is said to be* domain-consistent *if, for every accepting subset Q_i of the decomposition and every two states $q_1, q_2 \in Q_i$, q_1 and q_2 are domain-compatible.*

A Φ-deterministic 3DFA admits at least one closed and domain-consistent decomposition, the decomposition in which all subsets are singletons.

For C_1 as in Example 4.1, $D = \{\{q_0, q_1, dont\}, \{q_1, q_2, dont\}, \{q_3, dont\}, \{rej, dont\}\}$ is a closed and domain-consistent decomposition of $Q \cup \{rej\}$.

It will transpire that, in the construction of our minimal Φ-deterministic DFA, we will need closed and domain-consistent decompositions with minimum number of subsets of states. Therefore, it will be sufficient to consider decompositions for which there is only one rejecting subset (otherwise the rejecting subsets can be merged and a closed and domain-consistent decomposition with less number of subsets will be obtained) and there is no subset $Q_i = \{dont\}$ (such a subset can be merged with the rejecting subset and, again, a closed and domain-consistent decomposition with less number of subsets will be obtained). Consequently, we impose the following additional restrictions on the definition of a decomposition: (1) the decomposition contains only one rejecting subset and (2) there is no subset Q_i such that $Q_i = \{dont\}$.

Definition 5.6. *Let $C = (\Phi, Q \cup \{rej\}, h, q_0, Acc, \{rej\}, \{dont\})$ be a Φ-deterministic 3DFA and $D = \{Q_1, Q_2, \dots Q_n\}$ a closed and domain-consistent decomposition of $Q \cup \{rej\}$. Then we denote by C/D the set of all DFAs $M = (\Phi, Q' \cup \{rej'\}, h', q'_0, Q')$ such that*

- *the state set $Q' \cup \{rej'\}$ is D; the final states Q' are all the accepting subsets Q_i; rej' indicates the rejecting subset of D;*
- *the next-state function is defined by: for every $Q_i \in D$ and every $\phi \in \Phi$, if $h(Q_i, \phi) = \{dont\}$ then $h'(Q_i, \phi) = rej'$, otherwise $h'(Q_i, \phi) = Q_j$ for some $Q_j \in D$ such that $h(Q_i, \phi) \subseteq Q_j$;*
- *the initial state is some $Q_{i_0} \in D$ such that $q_0 \in Q_{i_0}$.*

The next-state function h' is well defined since the decomposition D is closed – see Note 5.1 and Definition 5.4.

The next-state function h' has the property that, for every $Q_i \in D$ and every $\phi \in \Phi$, $h'(Q_i, \phi) = Q_j$ for some $Q_j \in D$ such that $h(Q_i, \phi) \subseteq Q_j$. Additionally, when $h(Q_i, \phi) = \{dont\}$, Q_j is necessarily the rejecting subset indicated by rej'. This extra condition is imposed to ensure that the resulting M is Φ-deterministic. Indeed, let $\phi_1, \phi_2 \in \Phi$ and i, $1 \leq i \leq n$, such that $h'(Q_i, \phi_1) \in Q'$ and $h'(Q_i, \phi_2) \in Q'$. Then there exist $q_1 \in Q_i$ and $q_2 \in Q_i$ such that $h(q_1, \phi_1) \in Acc$ and $h(q_2, \phi_2) \in Acc$. Since D is domain-consistent, either $\phi_1 = \phi_2$ or $dom\ \phi_1 \cap dom\ \phi_2 = \emptyset$. Thus M is Φ-deterministic.

For C_1 as in Example 4.1, let $Q_1 = \{q_0, q_1, dont\}$, $Q_2 = \{q_1, q_2, dont\}$, $Q_3 = \{q_3, dont\}$, $Q_4 = \{rej, dont\}\}$. Then $D = \{Q_1, Q_2, Q_3, Q_4\}$ is a closed and domain-consistent decomposition of $Q \cup \{rej\}$. We observe that $h(Q_1, \phi_1) = \{q_1\} = Q_1 \cap Q_2$ and $h(Q_1, \phi_3) = \{q_1\} = Q_1 \cap Q_2$ and for all i and j, $1 \leq i \leq 3$, $1 \leq j \leq 4$, $(i,j) \notin \{(1,1), (1,3)\}$ there is precisely one k, $1 \leq k \leq 4$ such that $h(Q_i, \phi_j) \subseteq Q_k$. Then $C/D = \{M_1, M_2, M_3, M_4\}$, where $M_i = (\Phi, Q' \cup \{rej'\}, h'_i, q'_0, Q')$, $1 \leq i \leq 4$, with $Q' \cup \{rej'\} = D$, $q'_0 = Q_1$, $rej' = Q_4$ and the next-state functions defined as Figure 4 a), b), c) and d), respectively.

Theorem 5.1 (proved below) shows that a minimal Φ-deterministic DFA consistent with C can be found among the elements of C/D.

In the next lemmas the notations introduced in Definition 5.6 will be used.

Lemma 5.1. *M is consistent with C if and only if M covers C.*

PROOF. "$\Rightarrow$": Suppose M is consistent with C. Then $L_{C^-} \subseteq L_M \subseteq L_{C^+}$. We prove that q'_0 covers q_0. Let $p \in \Phi^*$. If $h(q_0, p) = rej$ then $p \in \overline{L_{C^+}}$. As $\overline{L_{C^+}} \subseteq \overline{L_M}$, it follows that $p \in \overline{L_M}$, so $h'(q'_0, p) = rej'$. If $h(q_0, p) \in Acc$ then

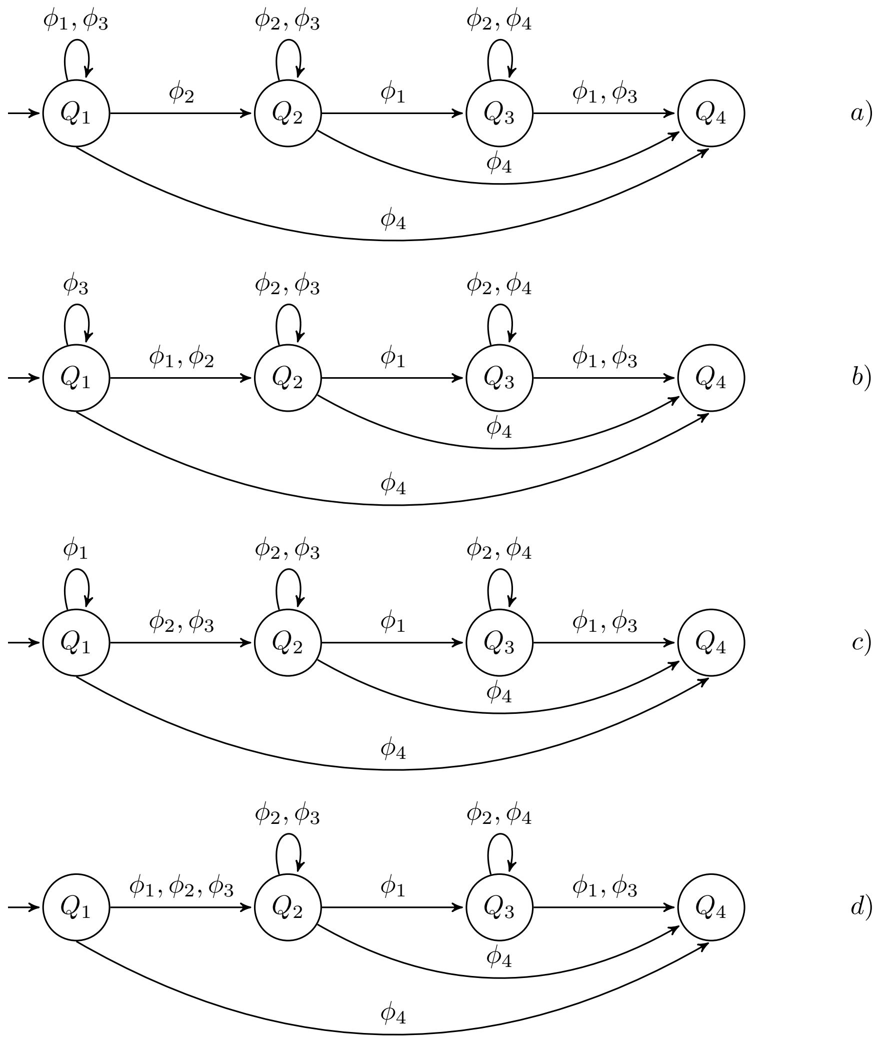

Figure 4: State transition diagram of M_1 (a), M_2 (b), M_3 (c) and M_4 (d)

$p \in L_{C^-}$. As $L_{C^-} \subseteq L_M$, it follows that $p \in L_M$, so $h'(q'_0, p) \in Q'$. Thus q'_0 covers q_0, so M covers C.

"$\Leftarrow$": Suppose M covers C. We prove that $L_{C^-} \subseteq L_M$. Let $p \in L_{C^-}$. Then $h(q_0, p) \in Acc$. As q'_0 covers q_0 it follows that $h(q'_0, p) \in Q'$, so $p \in L_M$. We prove now that $L_M \subseteq L_{C^+}$, which is equivalent to $\overline{L_{C^+}} \subseteq \overline{L_M}$. Let $p \in \overline{L_{C^+}}$. Then $h(q_0, p) = rej$. As q'_0 covers q_0, it follows that $h(q'_0, p) = rej'$, so $p \in \overline{L_M}$.

Lemma 5.2. *Let* $C = (\Phi, Q \cup \{rej\}, h, q_0, Acc, \{rej\}, \{dont\})$ *be a* Φ*-determi-*

nistic 3DFA and $D = \{Q_1, Q_2, \dots Q_n\}$ a closed and domain-consistent decomposition of $Q \cup \{rej\}$. Then for any $M \in C/D$, M covers C.

Proof. We prove that q_0' covers q_0. Let $p = \phi_1 \dots \phi_k \in \Phi^*$, $\phi_1, \dots, \phi_k \in \Phi$, $k \geq 0$. Let $h(q_j, \phi_{j+1}) = q_{j+1}$, $0 \leq j \leq k-1$. For any q_j, q_{j+1}, $0 \leq j \leq k-1$, as above, there exist $Q_{p_j}, Q_{p_{j+1}}$ from D, such that $q_j \in Q_{p_j}$ and $q_{j+1} \in Q_{p_{j+1}}$. From Definition 5.4 it follows that $h(Q_{p_j}, \phi_{j+1}) \subseteq Q_{p_{j+1}}$. Then $h'(Q_{p_j}, \phi_{j+1}) = Q_{p_{j+1}}$, $0 \leq j \leq k-1$. If $h(q_0, p) = rej$ then $q_k = rej \in Q_{p_k}$, so $Q_{p_k} = rej'$. Hence $h'(q_0', p) = rej'$. If $h(q_0, p) \in Acc$ then $q_k \in Acc$ and $q_k \in Q_{p_k}$, so Q_{p_k} is an accepting subset. Hence $h'(q_0', p) \in Q'$.

Lemma 5.3. *Let $C = (\Phi, Q \cup \{rej\}, h, q_0, Acc, \{rej\}, \{dont\})$ be a Φ-deterministic 3DFA and $D = \{Q_1, Q_2, \dots Q_n\}$ a closed and domain-consistent decomposition of $Q \cup \{rej\}$. Then for any $M \in C/D$, M is consistent with C.*

Proof. Follows from Lemma 5.1 and Lemma 5.2.

Theorem 5.1. *Let $C = (\Phi, Q \cup \{rej\}, h, q_0, Acc, \{rej\}, \{dont\})$ be a Φ–deterministic 3DFA. Then there exists $D = \{Q_1, Q_2, \dots Q_n\}$ a closed and domain-consistent decomposition of $Q \cup \{rej\}$ such that any $M \in C/D$ is a minimal Φ-deterministic DFA consistent with C.*

Proof. Let $M'' = (\Phi, Q'' \cup \{rej''\}, h'', q_0'', Q'')$ be a minimal Φ-deterministic DFA consistent with C. Let $\{q'' \in Q'' \cup \{rej''\} \mid \exists q \in Q$ such that q'' covers $q\} = \{q_1'', \dots, q_n''\}$. We build a closed and domain-consistent decomposition D and show that any $M \in C/D$ is a minimal Φ-deterministic DFA consistent with C.

Let $D = \{Q_1, Q_2, \dots Q_n\}$ with $Q_i = \{q \in Q \mid q_i''$ covers $q\}$, $1 \leq i \leq n$. By construction $Q_i \neq \emptyset$, $1 \leq i \leq n$. We prove that D is a closed and domain-consistent decomposition of $Q \cup \{rej\}$. We first prove that D is a decomposition. Let i, $1 \leq i \leq n$. If $q_i'' \in Q''$ then $q_i'' \neq rej''$. Let $q_i \in Q_i$. Since q_i'' covers q_i it follows that $q_i \neq rej$, so $q_i \in Q$. Thus $Q_i \subseteq Q$. If $q_i'' = rej''$ then $q_i'' \notin Q''$. Let $q_i \in Q_i$. Since q_i'' covers q_i it follows that $q_i \notin Acc$, so $q_i \in \{dont, rej\}$. Thus $Q_i \subseteq \{dont, rej\}$. We now prove that $Q_1 \cup Q_2 \cup \dots \cup Q_n = Q \cup \{rej\}$. Assume otherwise. Then there exists $q \in Q \cup \{rej\}$ such that $q \notin Q_1 \cup Q_2 \cup \dots \cup Q_n$. Then, for every $q'' \in Q'' \cup \{rej''\}$, q'' does not cover q. Let $p \in \Phi^*$ such that $h(q_0, p) = q$. Since q_0'' covers q_0, $h''(q_0'', p)$ covers q. This provides a contradiction and so $Q_1 \cup Q_2 \cup \dots \cup Q_n = Q \cup \{rej\}$. Then D is a decomposition. We prove that D is closed. Let i, $1 \leq i \leq n$, and $\phi \in \Phi$. Let $Q_i = \{q \in Q \mid q_i''$ covers $q\}$. Let $h''(q_i'', \phi) = q_j''$. Then for every $q_i \in Q_i$, q_j'' covers $h(q_i, \phi)$. Thus $h(Q_i, \phi) \subseteq Q_j = \{q \in Q \mid q_j''$ covers $q\}$. Hence D is closed. We prove that D is domain-consistent. Let i, $1 \leq i \leq n$, and $q_1, q_2 \in Q_i = \{q \in Q \mid q_i''$ covers $q\}$. Suppose $h(q_1, \phi_1) \in Acc$ and $h(q_2, \phi_2) \in Acc$. Since q_i'' covers q_1 and q_i'' covers q_1, $h''(q_i'', \phi_1) \in Q''$ and $h''(q_i'', \phi_1) \in Q''$. As M'' is Φ-deterministic, either $\phi_1 = \phi_2$ or $dom\ \phi_1 \cap dom\ \phi_2 = \emptyset$. Thus D is domain-consistent.

As D is a closed and domain-consistent decomposition of $Q \cup \{rej\}$, by construction, any $M \in C/D$ is a Φ-deterministic DFA. By Lemma 5.3, any $M \in$

C/D is consistent with C. Since D has at most the same number of elements as the number of states of M'', any $M \in C/D$ has at most the same number of states as the number of states of M''. Since M'' is a minimal Φ-deterministic DFA consistent with C, any $M \in C/D$ will have the same number of states as the number of states of M'' (hence M'' has n states and $Q_1, Q_2, \dots Q_n$ are distinct subsets). Thus, any $M \in C/D$ is a minimal Φ-deterministic DFA consistent with C.

6. Compatible states

As we have seen, a minimal Φ-deterministic DFA consistent with C can be found by checking the decompositions of $Q \cup \{rej\}$ of $1, 2, \dots$ elements until a closed and domain-consistent decomposition D is found. On the other hand, many decompositions can be outright eliminated using the concept of compatible pairs of states and Theorem 6.1 below.

Definition 6.1. *Let $q_1, q_2 \in Q \cup \{rej\}$; q_1 and q_2 are said to be* compatible *if there exists $D = \{Q_1, Q_2, \dots Q_n\}$ a closed and domain-consistent decomposition of $Q \cup \{rej\}$ such that $q_1, q_2 \in Q_i$ for some i, $1 \le i \le n$.*

Definition 6.2. *States $q_1, q_2 \in Q \cup \{rej\}$ are said to be* acceptance-compatible *if either: (1) $q_1, q_2 \in Q$ or (2) $q_1, q_2 \in \{dont, rej\}$.*

Theorem 6.1. *Let $q_1, q_2 \in Q \cup \{rej\}$. Then q_1 and q_2 are compatible if and only if for any $p \in \Phi^*$, $q_1' = h(q_1, p)$ and $q_2' = h(q_2, p)$ are acceptance-compatible and domain-compatible.*

Proof. "$\Rightarrow$": Suppose q_1 and q_2 are compatible. Then there exists $D = \{Q_1, Q_2, \dots Q_n\}$ a closed and domain-consistent decomposition of $Q \cup \{rej\}$ such that $q_1, q_2 \in Q_i$ for some i, $1 \le i \le n$. Let $p \in \Phi^*$, $h(q_1, p) = q_1'$ and $h(q_2, p) = q_2'$. Since D is a closed decomposition, there exists j, $1 \le j \le n$, such that $h(Q_i, \phi) \subseteq Q_j$. Then $q_1', q_2' \in Q_j$, so either $q_1', q_2' \in Q$ or $q_1', q_2' \in \{dont, rej\}$. Thus q_1' and q_2' are acceptance-compatible. Since D is a domain-consistent decomposition, q_1' and q_2' are domain-compatible.

"$\Leftarrow$": For every $p \in \Phi^*$, we define $R_p = \{h(q_1, p), h(q_2, p)\}$. Let $R = \bigcup_{p \in \Phi^*} R_p$. Let $D = \{R_p \mid p \in \Phi^*\} \cup \{\{q\} \mid q \in Q \cup \{rej\} \setminus R\}$. As $Q \cup \{rej\}$ is a finite set, there will be a finite number of sets R_p, so the number of elements of D will be finite. We denote them $Q_1, Q_2, \dots Q_n$. We prove that $D = \{Q_1, Q_2, \dots Q_n\}$ is a closed and domain-consistent decomposition of $Q \cup \{rej\}$. We first prove that D is a decomposition. Let i, $1 \le i \le n$. Then either $Q_i = R_\phi$ for some $p \in \Phi$ or $Q_i = \{q\}$ for some $q \in Q \cup \{rej\} \setminus R$. Suppose $Q_i = R_\phi$ for some $p \in \Phi$. Since $q_1' = h(q_1, p)$ and $q_2' = h(q_2, p)$ are acceptance-compatible either $q_1', q_2' \in Q$ or $q_1', q_2' \in \{dont, rej\}$. Thus $Q_i \subseteq Q$ or $Q_i \subseteq \{dont, rej\}$. If $Q_i = \{q\}$ for some $q \in Q \cup \{rej\} \setminus R$ then, clearly, $Q_i \subseteq Q$ or $Q_i \subseteq \{dont, rej\}$. $Q_1 \cup Q_2 \cup \dots \cup Q_n = Q \cup \{rej\}$ follows from the construction of D. Then D is a decomposition. We prove that D is closed. Let

i, $1 \leq i \leq n$. Suppose $Q_i = R_p$ for some $p \in \Phi^*$. Let $\phi \in \Phi$ and $R_{p\phi} = Q_j$, $1 \leq j \leq n$. Then $h(Q_i, \phi) \subseteq Q_j$. If $Q_i = \{q\}$ for some $q \in Q \cup \{rej\} \setminus R$ then, clearly, $h(Q_i, \phi) \subseteq Q_j$ for some j, $1 \leq j \leq n$. Thus D is closed. We prove that D is domain-consistent. Let i, $1 \leq i \leq n$. If Q_i contains two distinct states then $Q_i = R_p$ for some $p \in \Phi^*$ and the two states are $h(q_1, p) = q'_1$ and $h(q_2, p) = q'_2$. Then q'_1 and q'_2 are domain-compatible. Thus D is domain-consistent. Therefore D is a closed and domain-consistent decomposition of $Q \cup \{rej\}$. By definition $q_1, q_2 \in R_\epsilon$ (ϵ is the empty path) and so the implication follows.

On the basis of this result, we provide an algorithm for determining the pairs of compatible states, as described next. Let $Q \cup \{rej\} = \{q_1, q_2, \ldots q_l\}$. Let $PQ = \{(q_i, q_j) \mid 1 \leq i \leq l-1, i+1 \leq j \leq l\}$. The algorithm keeps a table as a mapping from PQ to $2^{PQ} \cup \{yes, no\}$, where $yes \neq no$ and $yes, no \notin 2^{PQ}$. For $q_i, q_j \in Q \cup \{rej\}$, $q_i \neq q_j$, we denote $next(q_i, q_j) = \{(q'_i, q'_j) \mid h(q_i, \phi) = q'_i, h(q_j, \phi) = q'_j, \phi \in \Phi, q'_i \neq q'_j, q'_i \neq dont, q'_j \neq dont\} \setminus \{(q_i, q_j)\}$.

The initial table $T_0 : PQ \longrightarrow 2^{PQ} \cup \{yes, no\}$ is defined by: $T_0(q_i, q_j) = yes$, if q_i and q_j are acceptance-compatible and domain-compatible and $next(q_i, q_j) = \emptyset$; $T_0(q_i, q_j) = no$, if q_i and q_j are not acceptance-compatible or not domain-compatible; $T_0(q_i, q_j) = next(q_i, q_j)$, if q_i and q_j are acceptance-compatible and domain-compatible and $next(q_i, q_j) \neq \emptyset$.

Suppose that we have constructed T_k, $k \geq 0$. Then T_{k+1} is defined by: $T_{k+1}(q_i, q_j) = yes$, if $T_k(q_i, q_j) = yes$; $T_{k+1}(q_i, q_j) = no$, if $T_k(q_i, q_j) = no$; $T_{k+1}(q_i, q_j) = no$, if $T_k(q_i, q_j) \notin \{yes, no\}$ and there exists $(q'_i, q'_j) \in T_k(q_i, q_j)$ such that $T_k(q'_i, q'_j) = no$ and $T_{k+1}(q_i, q_j) = T_k(q_i, q_j)$, otherwise. ($dont$ is compatible with any state in $Q \cup \{rej\}$ and that, for any accepting state $q \in Acc$, q and rej are not compatible, so there is no need to construct $T_t(q_i, q_j)$, $t \geq 0$, when $q_i \in \{dont, rej\}$ or $q_j \in \{dont, rej\}$.)

Since the set $Q \cup \{rej\}$ is finite, there exists $k \geq 0$ such that $T_k = T_{k+1}$. Also, for all $t \geq k$, $T_k = T_t$. Theorem 6.2 below provides a stopping criterion for the algorithm. The complexity of the algorithm is polynomial in the size of the state space and the number of processing functions of C.

Theorem 6.2. *Let k be the minimum index for which $T_k = T_{k+1}$. Then for every pair of states $q_i, q_j \in Q \cup \{rej\}$, $q_i \neq q_j$, q_i and q_j are compatible if and only $T_k(q_i, q_j) \neq no$.*

PROOF. "$\Rightarrow$": Suppose q_i and q_j are compatible. By Theorem 6.1, for any $p \in \Phi^*$, $q'_i = h(q_1, p)$ and $q'_j = h(q_j, p)$ are acceptance-compatible and domain-compatible. We prove by induction on $t \geq 0$ that $T_t(q_i, q_j) \neq no$. Since q_i and q_j are acceptance-compatible and domain-compatible, $T_0(q_i, q_j) \neq no$. Suppose $T_t(q_i, q_j) \neq no$. If $T_t(q_i, q_j) = yes$ then $T_{t+1}(q_i, q_j) = yes$. Otherwise $T_t(q_i, q_j) = next(q_i, q_j)$. Assume $T_{t+1}(q_i, q_j) = no$. Then there exists $(q'_i, q'_j) \in T_t(q_i, q_j)$ such that $T_t(q'_i, q'_j) = no$. Then there exists $\phi \in \Phi$ such that $h(q_i, \phi) = q'_i$ and $h(q_j, \phi) = q'_j$ are not acceptance-compatible or not domain-compatible. This provides a contradiction, as required. Thus $T_t(q_i, q_j) \neq no$ for all $t \geq 0$.

"$\Leftarrow$": Suppose $T_k(q_i, q_j) \neq no$. First, observe that q_1 and q_2 are acceptance-compatible and domain-compatible. Indeed, if we assume otherwise then $T_0(q_i, q_j)$ $\neq no$ and so $T_k(q_i, q_j) \neq no$, which provides a contradiction. We prove by induction on the length of p that for any $p \in \Phi^*$ (1) $q'_1 = h(q_1, p)$ and $q'_2 = h(q_2, p)$ are acceptance-compatible and domain-compatible and (2) $T_k(q'_i, q'_j) \neq no$. For $p = \epsilon$, $q'_1 = q_1$ and $q'_2 = q_2$. The statement follows since q_1 and q_2 are acceptance-compatible and domain-compatible and $T_k(q_i, q_j) \neq no$. Suppose that (1) $q'_1 = h(q_1, p)$ and $q'_2 = h(q_2, p)$ are acceptance-compatible and domain-compatible and (2) $T_k(q'_i, q'_j) \neq no$. Let $\phi \in \Phi$, $q''_1 = h(q'_1, p\phi)$ and $q''_2 = h(q'_2, p\phi)$. Suppose q''_i and q''_j are not acceptance-compatible or not domain-compatible. Then $T_0(q''_i, q''_j) = no$ and so $T_k(q''_i, q''_j) = no$. Then $T_{k+1}(q'_i, q'_j) = no$. Since $T_k = T_{k+1}$, $T_k(q'_i, q'_j) = no$. This provides a contraction and so q''_i and q''_j are acceptance-compatible and domain-compatible. Suppose $T_k(q''_i, q''_j) = no$. Then $T_{k+1}(q''_i, q''_j) = no$, so $T_k(q'_i, q'_j) = no$. This provides a contradiction, so $T_k(q'_i, q'_j) \neq no$. Then for any $p \in \Phi^*$, $q'_i = h(q_i, p)$ and $q'_j = h(q_j, p)$ are acceptance-compatible and domain-compatible. Then by Theorem 6.1, q_i and q_j are compatible.

Example 6.1. *Let $C = (\Phi, Q \cup \{rej\}, h, q_0, Acc, \{rej\}, \{dont\})$ be as follows. $\Phi = \{\phi_1, \phi_2, \phi_3, \phi_4, \phi_5\}$; for simplicity we omit the definitions of the processing functions, but we state instead that dom $\phi_i \cap$ dom $\phi_j = \emptyset$ for all i, j, $1 \leq i < j \leq 5$, $(i, j) \neq (4, 5)$ and dom $\phi_4 \cap$ dom $\phi_5 \neq \emptyset$; $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, dont\}$; h defined as in Figure 5. It can be observed that the pairs of states (q_i, q_j), $1 \leq i < j \leq 4$, are domain-compatible and the pairs of states (q_i, q_5), $1 \leq i \leq 4$ are not domain-compatible. T_0, T_1 and T_2 are as shown in Table 6. It can be observed that $T_2 = T_1$. Then the pairs of compatible states are: $(q_0, q_1), (q_1, q_2), (q_1, q_3), (q_2, q_3), (q_3, q_4)$ and $(q_i, dont)$, $0 \leq i \leq 5$, $(rej, dont)$.*

7. Constructing minimal Φ-deterministic DFA

In order to obtain the decompositions used in the construction of a minimal Φ-deterministic DFA consistent with the Φ-deterministic 3DFA C, it is useful to construct the maximal compatible set of states of C.

Definition 7.1. *A set of states $R \subseteq Q \cup \{rej\}$ is a set of* compatible *states if all states in R are pairwise compatible. R is called a* maximal compatible *set of states if there is no larger set of compatible states that contains R.*

The set MC of maximal compatible sets of accepting states can be gradually constructed by an algorithm that at each step, for each compatible pair of states q_i and q_j, adds state q_j to all subsets of sets in MC that contain states compatible with q_j; if no such subsets are found then the set $\{q_i, q_j\}$ is added to MC. For C as in the second running example,

$$MC = \{\{q_0, q_1\}, \{q_1, q_2, q_3\}, \{q_3, q_4\}, \{q_5\}\}.$$

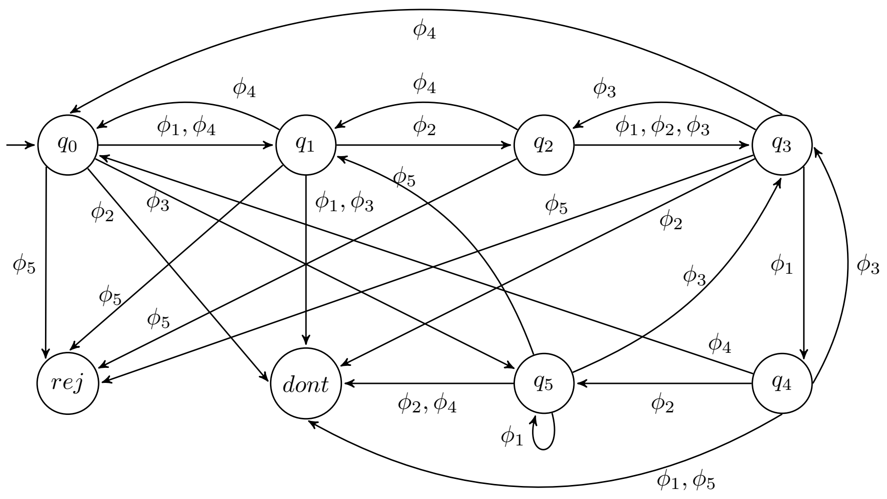

Figure 5: State transition diagram of C

The set of all maximal compatible sets of (accepting, rejecting and don't care) states of C is

$$\{\{q_0, q_1, dont\}, \{q_1, q_2, q_3, dont\}, \{q_3, q_4, dont\}, \{q_5, dont\}, \{dont, rej\}\}.$$

Once MC has been determined, we construct all decompositions (if any) $D = \{Q_1, Q_2, \ldots, Q_n\}$ of $Q \cup \{rej\}$ decompositions $D = \{Q_1, Q_2, \ldots, Q_n\}$, $n \geq 2$, for which there is precisely one i, $1 \leq i \leq n$, such that $Q_j = Q_j^a \cup \{dont\}$ for all j, $1 \leq j \leq n-1$, with Q_j^a contained in a maximal compatible set of accepting states such that $\bigcup_{1 \leq j \leq n, j \neq i} Q_j^a = Acc$ and $Q_n = \{rej, dont\}$. The value of n is then increased until a closed decomposition is found. The any element of C/D is a minimal Φ-deterministic DFA consistent with C, Since the set of all maximal compatible sets of states of C is closed, the algorithm will find a solution for a value of n that does not exceed the number of elements of MC plus 1.

For the second running example the algorithm will return $n = 5$ elements. There is more than one decomposition that satisfy the above requirement. One is $D = \{Q_1, Q_2, Q_3, Q_4, Q_5\}$, $Q_1 = \{q_0, q_1, dont\}$, $Q_2 = \{q_1, q_2, q_3, dont\}$, $Q_3 = \{q_3, q_4, dont\}$, $Q_4 = \{q_5, dont\}$, $Q_5 = \{dont, rej\}$. C/D contains more than one DFA. One is $M' = (\Phi, Q' \cup \{rej'\}, h', q_0', Q')$ with $Q' = \{Q_1, Q_2, Q_3, Q_4\}$, $rej' = Q_5$, $q_0' = Q_1$ and h' defined is in Figure 6.

The above algorithm involves some enumeration and so, naturally, is computationally expensive. The following heuristic, inspired from automata applications [29], can be used to significantly reduce its complexity. Instead of constructing a minimal Φ-deterministic DFA consistent with C, the following

Table 1: T_0, T_1, T_2

T_0	q_1	q_2	q_3	q_4	q_5
q_0	*yes*	$\{(q_1,q_3),(q_3,q_5)\}$	$\{(q_1,q_4),(q_2,q_5),(q_0,q_1)\}$	$\{(q_3,q_5),(q_0,q_1)\}$	*no*
q_1		$\{(q_2,q_3),(q_0,q_1)\}$	*yes*	$\{(q_2,q_5)\}$	*no*
q_2			$\{(q_3,q_4),(q_0,q_1)\}$	$\{(q_3,q_5),(q_0,q_1)\}$	*no*
q_3				$\{(q_2,q_3)\}$	*no*
q_4					*no*

T_1	q_1	q_2	q_3	q_4	q_5
q_0	*yes*	*no*	*no*	*no*	*no*
q_1		$\{(q_2,q_3),(q_0,q_1)\}$	*yes*	*no*	*no*
q_2			$\{(q_3,q_4),(q_0,q_1)\}$	*no*	*no*
q_3				$\{(q_2,q_3)\}$	*no*
q_4					*no*

T_2	q_1	q_2	q_3	q_4	q_5
q_0	*yes*	*no*	*no*	*no*	*no*
q_1		$\{(q_2,q_3),(q_0,q_1)\}$	*yes*	*no*	*no*
q_2			$\{(q_3,q_4),(q_0,q_1)\}$	*no*	*no*
q_3				$\{(q_2,q_3)\}$	*no*
q_4					*no*

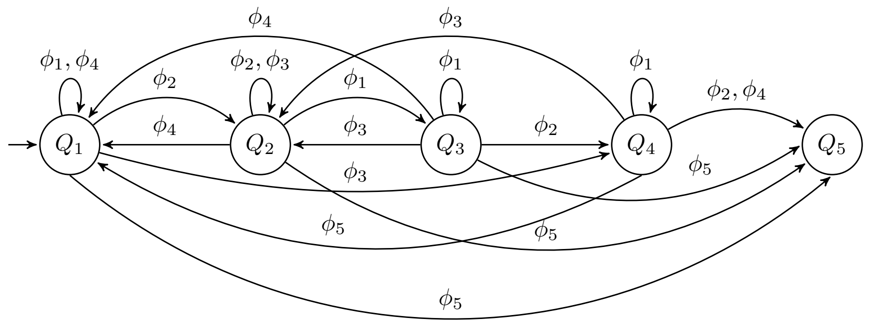

Figure 6: State transition diagram of M'

Φ-deterministic DFA consistent with C, which is not necessarily minimal, is constructed.

Definition 7.2. *Let $Q_1, Q_2, \dots Q_n$ be the maximal compatible sets of states of $C = (\Phi, Q \cup \{rej\}, h, q_0, Acc, \{rej\}, \{dont\})$. We define a DFA $M'' = (\Phi, Q'' \cup$*

$\{rej''\}, h'', q_0'', Q'')$ as follows:

- *the states set is $Q'' \cup \{rej''\} = \{Q_1, Q_2, \dots, Q_n\}$; Q'' is the set of accepting subsets of D; rej'' is the rejecting subset;*
- *the next-state function is defined by: for every $Q_i \in D$ and every $\phi \in \Phi$, if $h(Q_i, \phi) = \{dont\}$ then $h''(Q_i, \phi) = rej''$, otherwise $h''(Q_i, \phi) = R$ where R is the largest subset Q_j such that $h(Q_i, \phi) \subseteq Q_j$ (if more than one such subset exist, one is randomly chosen);*
- *the initial state is the largest subset that contains q_0 (if more than one such subset exist, one is randomly chosen);*
- *the final states are all the accepting subsets Q_i.*

Clearly, $D = \{Q_1, Q_2, \dots Q_n\}$ is a decomposition of $Q \cup \{rej\}$. M'' is well defined since the next states of a set of compatible states are also compatible states. Naturally, there may exist more than one such M''.

For the second running example, $Q_1 = \{q_0, q_1, dont\}$, $Q_2 = \{q_1, q_2, q_3, dont\}$, $Q_3 = \{q_3, q_4, dont\}$, $Q_4 = \{q_5, dont\}$ and $Q_5 = \{dont, rej\}$ are the maximal compatible sets of states of C as defined in our second running example. Then $M'' = (\Phi, Q'' \cup \{rej''\}, h'', q_0'', Q'')$, where $Q'' = \{Q_1, Q_2, Q_3, Q_4\}$, $rej'' = Q_5$, $q_0'' = Q_1$ and h' as in Figure 7. It can be observed that, in this case, M'' is uniquely determined.

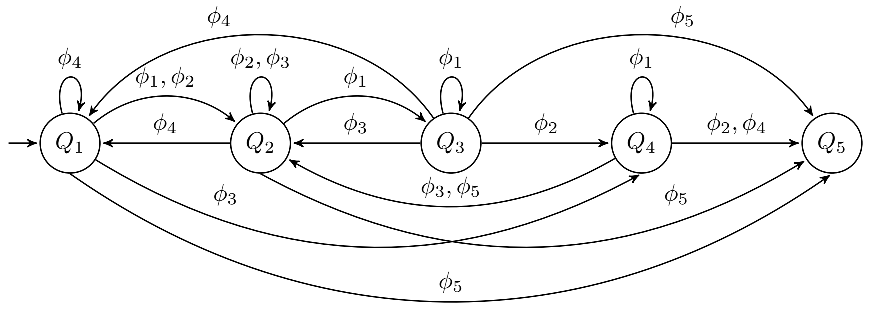

Figure 7: State transition diagram of M''

According to our experimental results, although M'' (as defined above) is not necessarily minimal, it is usually close to a minimal Φ-deterministic DFA and its construction is considerably faster than our original algorithm. However, an in-depth investigation of the results produced by this heuristic is beyond the scope of this paper.

If this heuristic is used then the overall algorithm reduces to (1) determining the pairs of compatible states of the 3DFA C, (2) constructing the set of maximal compatible sets of C and (3) constructing M''. Thus, in this case, the complexity of the algorithm is polynomial in the size of the the state space and the number of processing functions of C.

8. Conclusions

In this paper we have investigated some fundamental problems and algorithms related to constructing a minimal deterministic stream X-machine Z consistent with a deterministic 3DFA that has as inputs the processing functions of Z and a heuristic for improving the complexity of the solution to this problem. This theoretical problem is essential for learning deterministic stream X-machines, special classes of extended finite state machines, from queries. In practice, one might encounter applications, such as agent-based systems using the FLAME framework, or state-based testing strategies [34] for classes of membrane systems used in various development projects, requesting more complex models than simple stream X-machines. Consequently, an important continuation of this work is the extension of the current results to *communicating* stream X-machines. Due to the established connections between X-machines and membrane systems, as mentioned in the Introduction section, it is also expected that this paper will open the door for algebraic inference approaches for membrane systems.

Acknowledgements. The authors would like to thank the anonymous reviewers for carefully reading the paper and making suggestions that led to its improvement.

[1] G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, Springer Verlag, Berlin, Heidelberg, 1997.

[2] J. E. Hopcroft, J. D. Ullman, An Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.

[3] B. Krena, T. Vojnar, Automated formal analysis and verification: an overview, Int. J. General Systems 42 (4) (2013) 335–365. doi:10.1080/03081079.2012.757437.
URL `https://doi.org/10.1080/03081079.2012.757437`

[4] A. P. Mathur, Foundations of Software Testing (2nd Edition), Addison-Wesley, 2014.

[5] D. Angluin, Learning regular sets from queries and counterexamples, Inf. Comput. 75 (2) (1987) 87–106. doi:10.1016/0890-5401(87)90052-6.
URL `https://doi.org/10.1016/0890-5401(87)90052-6`

[6] H. Hungar, O. Niese, B. Steffen, Domain-specific optimization in automata learning, in: CAV, 2003, pp. 315–327. doi:10.1007/978-3-540-45069-6_31.
URL `https://doi.org/10.1007/978-3-540-45069-6_31`

[7] T. Berg, B. Jonsson, H. Raffelt, Regular inference for state machines with parameters, in: FASE, 2006, pp. 107–121. doi:10.1007/11693017_10.
URL `https://doi.org/10.1007/11693017_10`

[8] F. Ipate, Learning finite cover automata from queries, J. Comput. Syst. Sci. 78 (1) (2012) 221–244. doi:10.1016/j.jcss.2011.04.002.
URL `https://doi.org/10.1016/j.jcss.2011.04.002`

[9] F. Howar, B. Steffen, Active automata learning in practice - An annotated bibliography of the years 2011 to 2016, in: Machine Learning for Dynamic Software Analysis, Lecture Notes in Computer Science, Vol. 11025, Springer Verlag, 2018, pp. 123–148. doi:10.1007/978-3-319-96562-8_5.
URL `https://doi.org/10.1007/978-3-319-96562-8_5`

[10] I. Dinca, F. Ipate, A. Stefanescu, Model learning and test generation for Event-B decomposition, in: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I, 2012, pp. 539–553. doi:10.1007/978-3-642-34026-0_40.
URL `https://doi.org/10.1007/978-3-642-34026-0_40`

[11] M. Fantinato, M. Jino, Applying extended finite state machines in software testing of interactive systems, in: DSV-IS 2003: Proceedings of 10th International Workshop on Interactive Systems. Design, Specification, and Verification, Lecture Notes in Computer Science, vol 2844, Springer Verlag, 2003, pp. 34–45. doi:10.1007/978-3-540-39929-2_3.
URL `https://doi.org/10.1007/978-3-540-39929-2_3`

[12] R. Yang, Z. Chen, Z. Zhang, B. Xu, EFSM-based test case generation: Sequence, data, and oracle, International Journal of Software Engineering and Knowledge Engineering 25 (4) (2015) 633–668. doi:10.1142/S0218194015300018.
URL `https://doi.org/10.1142/S0218194015300018`

[13] M. Holcombe, F. Ipate, Correct systems - building a business process solution, Applied computing, Springer, 1998.

[14] K. Bogdanov, M. Holcombe, F. Ipate, L. Seed, S. K. Vanak, Testing methods for X-machines: a review, Formal Asp. Comput. 18 (1) (2006) 3–30. doi:10.1007/s00165-005-0085-6.
URL `https://doi.org/10.1007/s00165-005-0085-6`

[15] R. M. Hierons, Checking experiments for stream X-machines, Theor. Comput. Sci. 411 (37) (2010) 3372–3385. doi:10.1016/j.tcs.2010.05.014.
URL `https://doi.org/10.1016/j.tcs.2010.05.014`

[16] M. G. Merayo, M. Núñez, R. M. Hierons, Testing timed systems modeled by stream X-machines, Software and System Modeling 10 (2) (2011) 201–217. doi:10.1007/s10270-009-0126-3.
URL `https://doi.org/10.1007/s10270-009-0126-3`

[17] R. M. Hierons, F. Ipate, Testing a deterministic implementation against a non-controllable non-deterministic stream X-machine, Formal Asp. Comput. 20 (6) (2008) 597–617. doi:10.1007/s00165-008-0087-2.
URL `https://doi.org/10.1007/s00165-008-0087-2`

[18] F. Ipate, Testing against a non-controllable stream X-machine using state counting, Theor. Comput. Sci. 353 (1-3) (2006) 291–316. doi:10.1016/j.tcs.2005.12.002.
URL `https://doi.org/10.1016/j.tcs.2005.12.002`

[19] F. Ipate, M. Holcombe, Testing data processing-oriented systems from stream X-machine models, Theor. Comput. Sci. 403 (2-3) (2008) 176–191. doi:10.1016/j.tcs.2008.02.045.
URL `https://doi.org/10.1016/j.tcs.2008.02.045`

[20] F. Ipate, D. Dranidis, A unified integration and component testing approach from deterministic stream X-machine specifications, Formal Asp. Comput. 28 (1) (2016) 1–20. doi:10.1007/s00165-015-0345-z.
URL `https://doi.org/10.1007/s00165-015-0345-z`

[21] A. Guignard, J. Faure, G. Faraut, Model-based testing of PLC programs with appropriate conformance relations, IEEE Trans. Industrial Informatics 14 (1) (2018) 350–359. doi:10.1109/TII.2017.2695370.
URL `https://doi.org/10.1109/TII.2017.2695370`

[22] D. E. Jackson, M. Holcombe, F. L. W. Ratnieks, Trail geometry gives polarity to ant foraging networks, Nature 432 (7019) (2004) 9079. doi:10.1038/nature03105.
URL `https://doi.org/10.1038/nature03105`

[23] R. H. Smallwood, M. Holcombe, The epitheliome project: multiscale agent-based modeling of epithelial cells, in: Proceedings of the 2006 IEEE International Symposium on Biomedical Imaging: From Nano to Macro, Arlington, VA, USA, 6-9 April 2006, 2006, pp. 816–819. doi:10.1109/ISBI.2006.1625043.
URL `https://doi.org/10.1109/ISBI.2006.1625043`

[24] Flame web site, `http://flame.ac.uk/`.

[25] Păun, Gh., Computing with membranes, Journal of Computer and System Sciences 61 (1) (2000) 108–143. doi:10.1006/jcss.1999.1693.
URL `https://doi.org/10.1006/jcss.1999.1693`

[26] Păun, Gh., G. Rozenberg, A. Salomaa (Eds.), The Oxford Handbook of Membrane Computing, Oxford University Press, Inc., 2010.

[27] J. Aguado, T. Bălănescu, A. Cowling, M. Gheorghe, M. Holcombe, F. Ipate, P systems with replicated rewriting and stream X-machines (Eilenberg machines), Fundam. Inform. 49 (1-3) (2002) 17–33.

URL http://content.iospress.com/articles/fundamenta-informaticae/fi49-1-3-03

[28] P. Kefalas, I. Stamatopoulou, I. Sakellariou, G. Eleftherakis, Transforming communicating X-machines into P systems, Natural Computing 8 (4) (2009) 816–832. doi:10.1007/s11047-008-9103-y.
URL https://doi.org/10.1007/s11047-008-9103-y

[29] Y. Chen, A. Farzan, E. M. Clarke, Y. Tsay, B. Wang, Learning minimal separating DFAs for compositional verification, in: Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings, 2009, pp. 31–45. doi:10.1007/978-3-642-00768-2_3.
URL https://doi.org/10.1007/978-3-642-00768-2_3

[30] O. Grinchtein, M. Leucker, N. Piterman, Inferring network invariants automatically, in: Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings, 2006, pp. 483–497. doi:10.1007/11814771_40.
URL https://doi.org/10.1007/11814771_40

[31] A. Gupta, K. L. McMillan, Z. Fu, Automated assumption generation for compositional verification, Formal Methods in System Design 32 (3) (2008) 285–301. doi:10.1007/s10703-008-0050-0.
URL https://doi.org/10.1007/s10703-008-0050-0

[32] J. M. Pena, A. L. Oliveira, A new algorithm for exact reduction of incompletely specified finite state machines, IEEE Trans. on CAD of Integrated Circuits and Systems 18 (11) (1999) 1619–1632. doi:10.1109/43.806807.
URL https://doi.org/10.1109/43.806807

[33] M. C. Paull, S. H. Unger, Minimizing the number of states in incompletely specified sequential switching functions, IRE Trans. Electronic Computers 8 (3) (1959) 356–367. doi:10.1109/TEC.1959.5222697.
URL https://doi.org/10.1109/TEC.1959.5222697

[34] M. Gheorghe, F. Ipate, S. Konur, Testing based on identifiable P systems using cover automata and X-machines, Inf. Sci. 372 (2016) 565–578. doi:10.1016/j.ins.2016.08.028.
URL https://doi.org/10.1016/j.ins.2016.08.028