Learning and Integration of Parameterized Components through Testing

Muzammil Shahbaz[1], Keqin Li[2], and Roland Groz[2]

[1] France Telecom R&D
Meylan, France.
muhammad.muzammilshahbaz@orange-ftgroup.com
[2] LIG, Computer Science Lab
Grenoble Universités, France
{Keqin.Li,Roland.Groz}@imag.fr

Abstract. We investigate the use of parameterized state machine models to drive integration testing, in the case where the models of components are not available beforehand. Therefore, observations from tests are used to learn partial models of components, from which further tests can be derived for integration. We have extended previous algorithms to the case of finite state models with predicates on input parameters and observable non-determinism. We also propose a new strategy where integration tests can be derived from the data collected during the learning process. Our work typically addresses the problem of assembling telecommunication services from black box COTS.

1 Introduction

Model based testing has gained momentum in many industrial fields, in particular in the domain of testing complex systems, e.g., telecom services, which are composed of various components developed independently. It is not uncommon for these components to be collected from different sources as COTS (Commercial-off-the-shelf), their formal models are not always available and no detailed technical corpora is provided with the components. Therefore, engineers find difficulty in providing a required system integration if they have limited knowledge of the behaviors of the components, which they use in the system.

To address this problem, we propose to generate formal models directly from the components through testing. These models are generated as state machine models so that rigorous techniques of model based integration testing could readily be applied. This provides us room to investigate methods of state machine inference from black box components, i.e., the components whose internal structure is unknown. Among various such methods, Angluin's algorithm [1] is well-known that learns a deterministic automata in a polynomial time. This work has yielded positive results in applied research [14], [7] etc., where real problems were put under case-studies. However there remained less explicit emphasis in these works on learning expressive models.

In our particular case of integration of a variety of components, we have observed that the nature of components integration elicit potential interoperability problems due to exchange of data values from arbitrarily complex domains. In this case, learning DFA models for such components would be inadequate and impractical due to the chance of state-explosion and loss of genericity of the model. Therefore, we need to advance from simple state-machine inference to the inference of more expressive models that can maintain the fine granularity of complex systems, i.e., parametric details and also some notion of nondeterminism. Also, as the size of input data is directly proportional to the testing effort, there is a good argument to model expressive forms that can detail the intended behaviors of the component in a compact form and can be learnt through less number of test cases. We have proposed techniques based on Angluin's algorithm to adapt it for more expressive models than DFA, starting from Mealy machines [9] to simple parameterized models [10].

In this paper, we enrich our model to incorporate parameterized predicates on transitions with observable nondeterminism. This model is more expressive compared to the models proposed in the previous works of automata inference [1], [7], [9], [10], [2] in terms of parameterized inputs/outputs, infinite domain of parameter values, predicates on input parameters and observable nondeterminism when interacting with input parameter values. Compared to usual EFSM models [13], [12], we stop short of including variables in the model, because when we learn a black box, we cannot distinguish in its internal structure what would be encoded as (control) state and what would be encoded in variables. All state information in our model is encoded in the state machine structure.

We propose an algorithm to infer such parameterized models based on Angluin's algorithm. We also have significantly improved the algorithm in two ways. The basic algorithm and all its adaptations stated above check for certain concepts in order to make a conjecture of the model. Inspired by [14], we reduced one of these concepts, called *consistency* and hence reduced the number of test cases needed to perform this concept. Furthermore, the algorithm assumes an oracle that provides a counterexample when the conjecture is wrong. In the context of industrial applications where this oracle assumption is quite unrealistic, we propose a technique to find potential counterexamples from the models taking advantage of our integration testing strategy. The counterexamples are provided back to the learning procedure to refine the learned model, thus making it an iterative process [9]. We also consider former approaches, e.g., property-based testing [11] and scenario-based testing [10] and propose a new integration testing technique which is illustrated with the help of an example of integrating two parameterized components. The organization of the paper is as follows. The formal definition of the parameterized model is given in section 2 and its learning algorithm is described in section 3. The integration testing strategy and related discussion is covered in section 4 and finally section 5 concludes the paper.

2 Parameterized Model

A *Parameterized Finite State Machine (PFSM) M* is a tuple $M = (Q, I, O, D_I, D_O, \Gamma, q_0)$, where

- Q is a finite set of states
- I is a finite set of input symbols
- O is a finite set of output symbols
- D_I is a set of input parameter values
- D_O is a set of output parameter values
- q_0 is an initial state
- Γ is a set of transitions

A transition $t \in \Gamma$ is described as: $t = (q, q', i, o, p, f)$, where $q \in Q$ is a source state, $q' \in Q$ is a target state, $i \in I$ is an input symbol, $o \in O$ is an output symbol, $p \subseteq D_I$ is a predicate on input parameter values and $f : p \longrightarrow D_O$ is an output parameter function. We consider that the model is restricted with the following three properties.

Property 1 (Input Enabled). The model is *input enabled*, i.e., $\forall q \in Q$, $\forall i \in I$ and $\forall x \in D_I$, $\exists t \in \Gamma$ such that $t = (q, q', i, o, p, f)$, in which $x \in p$.

The machine can be made input enabled by adding loop back transitions on a state for all those inputs (and associated predicate for parameter values) which are not acceptable for that state. Such transitions contain a special symbol Ω in O. Similarly, there exists transitions which do not take input parameter values into account. Such transitions contain a special symbol $\perp$ with the input symbol that expresses the absence of parameter value. For the sake of simplicity, we do not write this symbol while modeling a problem with PFSM.

Property 2 (Input Deterministic). The model is *input deterministic*, i.e., for $t_1, t_2 \in \Gamma$ such that $t_1 = (q_1, q_1', i_1, o_1, p_1, f_1)$, $t_2 = (q_2, q_2', i_2, o_2, p_2, f_2)$ and $t_1 \neq t_2$, if $q_1 = q_2 \wedge i_1 = i_2$ then $p_1 \cap p_2 = \phi$.

Property 3 (Observable). The model is *observable*, i.e., for $t_1, t_2 \in \Gamma$ such that $t_1 = (q_1, q_1', i_1, o_1, p_1, f_1)$, $t_2 = (q_2, q_2', i_2, o_2, p_2, f_2)$ and $t_1 \neq t_2$, if $q_1 = q_2 \wedge i_1 = i_2$ then $o_1 \neq o_2$.

Property 3 ensures that two transitions having same source state and same input symbol would generate different output symbols. This helps us determining the target states that are possibly different for each transition in the learning algorithm.

When M is in state $q \in Q$ and receives an input $i \in I$ along with the parameter value $x \in D_I$, then the target state q', the output o and the output parameter value function f are determined by the functions δ, λ and σ respectively, which are described as follows:

- $\delta : Q \times I \times D_I \longrightarrow Q$ is a target state function

– $\lambda : Q \times I \times D_I \longrightarrow O$ is an output function
– $\sigma : Q \times I \longrightarrow {D_O}^{D_I}$ is an output parameter function. ${D_O}^{D_I}$ is the set of all functions from D_I to D_O.

The properties 1 and 2 ensure that δ and λ are mappings. For an input symbol sequence $\omega = i_1, \ldots, i_k$ and an input parameter value sequence $\alpha = x_1, \ldots, x_k$, where each $i_j \in I, x_j \in D_I, 1 \leq j \leq k$, we define a parameterized input sequence, i.e., the association of ω and α as $\omega \otimes \alpha = i_1(x_1), ..., i_k(x_k)$, where each x_j is associated with i_j and $|\omega| = |\alpha|$. The association of output symbol sequence and output parameter value sequence is defined analogously. Then, for the state $q_1 \in Q$, when applying a parameterized input sequence $\omega \otimes \alpha$, M moves successively from q_1 to the states $q_{j+1} = \delta(q_j, i_j, x_j), \forall 1 \leq j \leq k$. We extend the functions from input symbols to parameterized input sequences as $\delta(q_1, \omega, \alpha) = q_{k+1}$ to denote the final state q_{k+1} and $\lambda(q_1, \omega, \alpha) = o_1(y_1), ..., o_k(y_k)$, where each $o_j = \lambda(q_j, i_j, x_j), y_j = \sigma(q_j, i_j)(x_j), \forall 1 \leq j \leq k$, to denote the complete parameterized output sequence, when applying $\omega \otimes \alpha$ on q_1.

An example of PFSM model is given in Figure 1, in which $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$, $I = \{a, u\}$, $O = \{s, t\}$, $D_I = D_O = \mathbb{Z}$, the set of integers.

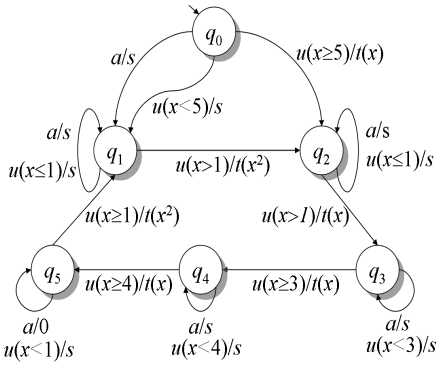

Fig. 1. Example of PFSM Model

3 Algorithm

Assume an unknown PFSM $M = (Q, I, O, D_I, D_O, \Gamma, q_0)$ with known input symbols I and input parameter domain D_I can be used to model a component C. For any parameterized input sequence or a test case $\omega \otimes \alpha, (\omega \in I^*, \alpha \in D_I^*)$ for a component, we assume that $\lambda(q_0, \omega, \alpha)$ can be known from testing. We also assume that C can be reset to its initial state before each test. The key part of the learning algorithm is using observation table. We define the structure of the table and related definitions in the section below and then the algorithm in the subsequent section.

3.1 Observation Table

The observation table is used to generate test cases for an unknown component, to organize the result of each test case and finally to make a PFSM conjecture when certain properties on the table are satisfied. The rows and columns of the table are labelled by input strings which are associated with the input parameter values in order to construct test cases. The result of a test case is organized in the cells of the table in the form of a pair of input parameter value sequence and parameterized output. After the table conforms to the properties, a conjecture is made where some rows of the table are regarded as states and transitions are derived from the observations recorded in the table. We shall describe the basic structure and properties of the table in this section and rest of the explanation regarding construction of test cases, organization of outputs and making a conjecture out of the table will be explained in the next section.

Structure Let $\mathcal{U} = \{\omega \otimes \alpha | \omega \in I^+, \alpha \in {D_I}^+\} \cup \{\omega | \omega \in I^*\}$ be the set of parameterized input sequences and input symbol sequences. We define $IS(u), u \in \mathcal{U}$, an input symbol sequence from u such that if $u = \omega$ or $u = \omega \otimes \alpha, \omega \in I^*, \alpha \in {D_I}^*$, then $IS(u) = \omega$. Also, $pref(\gamma)^k$ is the prefix of some sequence $\gamma \in I^* \cup O^* \cup {D_I}^* \cup {D_O}^*$ of length $k \leq |\gamma|$. For example, for $\gamma = i_1, ..., i_n$, where every $i_j \in I, 1 \leq j \leq n, pref(\gamma)^k = i_1, ..., i_k, 1 \leq k \leq n$. Similarly, $suff(\gamma)^k$ is the suffix of γ of length $k \leq |\gamma|$. Let $\mathcal{P} = \{(\alpha, \varpi \otimes \beta) | \alpha \in {D_I}^+, \varpi \in O^+, \beta \in {D_O}^+\}$ be the set of the pair of input parameter value sequence and parameterized output sequence.

		E	
		a	u
S	ϵ	$(\perp, s \otimes \perp)$	$(1, s \otimes \perp)$, $(5, t \otimes 5)$
R	a	$((\perp, \perp), s \otimes \perp)$	$((\perp, 1), s \otimes \perp)$, $((\perp, 5), t \otimes 25)$
	u	$((1, \perp), s \otimes \perp)$	$((5, 1), s \otimes \perp)$, $((1, 5), t \otimes 25)$

Table 1. Example of an Observation Table

An observation table is denoted as (S, R, E, T). S and R are nonempty finite sets of input strings and make the rows of the table. *S is used to identify potential states in the conjecture* and R is used to satisfy properties on the table. In PFSM, states are not only determined by input symbol sequence but also by parameter value sequence, thus formally, $S \subseteq \mathcal{U}$ is a set of input symbol sequences and parameterized input sequences, and $R \subseteq \mathcal{U}$ extends the rows such that for all $r \in R$, there exists $s \in S, e \in E$ and $IS(r) = IS(s) \cdot e$. Whenever, a sequence s is added to S, R is extended in the following way: i) if $s = \omega, \omega \in I^*$ an input symbol sequence, then R will be extended by $\omega \cdot i$, for all $i \in I$. ii) if $s = \omega \otimes \alpha, \omega \in I^+, \alpha \in {D_I}^+$ a parameterized input sequence, then R will be extended by $\omega \cdot i \otimes \alpha \cdot x$, for all $i \in I$, where x is selected from D_I. The selection policy for parameter values is up to the system's specific requirements, however a general idea is given in section 4.

$E \subseteq I^+$ is a nonempty finite set of input symbol sequences that make the columns of the table and *separate the different states of the conjecture.* The elements of $(S \cup R) \times E$ are used to construct test cases in the algorithm which are associated with parameter value sequences from ${D_I}^+$, and their results (observations) are organized in the table with the help of a function T mapping from $(S \cup R) \times E$ to $2^{\mathcal{P}}$. For example, a parameterized output of a test case derived from $s \in S \cup R$, $e \in E$ and associated with parameter value sequence $\alpha \in {D_I}^+$ will be organized in $T(s, e)$ in the form of a pair of input parameter value sequence and parameterized output sequence, i.e., $(\alpha, \varpi \otimes \beta), \alpha \in {D_I}^+, \varpi \in O^+, \beta \in {D_O}^+$. The observations from $T(s, e)$ are used to *identify potential transitions in the conjecture and label them with input/output and parameter values.* Table 1 is an example of an observation table, in which S contains only one row ϵ and R contains two rows a and u, whereas E contains two columns a and u respectively.

Properties Each test case driven from $s \in S \cup R, e \in E$ may generate different parameterized output sequences depending upon the selection of different input parameter value sequences from ${D_I}^+$. Thus, there may exist $(\alpha_1, \varpi_1 \otimes \beta_1), (\alpha_2, \varpi_2 \otimes \beta_2) \in T(s, e)$ such that $\varpi_1 \neq \varpi_2$, i.e, $T(s, e)$ contains pairs in which the output sequences are different. Let $\eta(T(s, e))$ be the number of different output sequences contained by $T(s, e)$, then we can divide $T(s, e)$ into $\eta(T(s, e))$ *distinguishing subsets*, i.e., $T(s, e) = \bigcup_{k=1}^{\eta(T(s,e))} d_k(s, e)$, where in each $d_k(s, e) = \{({\alpha_1}^{(k)}, {\varpi_1}^{(k)} \otimes {\beta_1}^{(k)}), \ldots, ({\alpha_m}^{(k)}, {\varpi_m}^{(k)} \otimes {\beta_m}^{(k)})\} \subseteq T(s, e), m = |d_k(s, e)|$, $\varpi_1 = \ldots = \varpi_m$, the output sequences are same. Let $OS(d_k(s, e)) = {\varpi_1}^{(k)} = \ldots = {\varpi_m}^{(k)}$ be the output sequence from $d_k(s, e)$, then for any $d_1(s, e), d_2(s, e) \subset T(s, e)$, $OS(d_1(s, e)) \neq OS(d_2(s, e))$ and $d_1(s, e) \cap d_2(s, e) = \emptyset$. For every $d_k(s, e)$, we define $\rho(d_k(s, e)) = \{{\alpha_1}^{(k)}, \ldots, {\alpha_m}^{(k)}\}$, a set of *distinguishing parameter value sequence* from $d_k(s, e)$, and $PS(d_k(s, e)) = \{(suff({\alpha_1}^{(k)})^{|e|}, {\beta_1}^{(k)}), \ldots, (suff({\alpha_m}^{(k)})^{|e|}, {\beta_m}^{(k)})\}$, the set of pairs of i/o parameter value sequences from $d_k(s, e)$.

Since $T(s, e), s \in S \cup R, e \in E$ represents a possible transition in the conjecture, if $T(s, e)$ contains many distinguishing subsets then each subset may represent a different transition. Therefore, we call such s a disputed row. Formally, $s \in S$ is *disputed* iff for any $e \in E$, $\eta(T(s, e)) > 1$, i.e., $T(s, e)$ contains more than one *distinguishing subsets*. The table must contain additional rows to treat disputed rows. A disputed row s is *treated* iff for every distinguishing subset $d_k(s, e) \subset T(s, e), 1 \leq k \leq \eta(T(s, e))$, there exists $t \in S \cup R$ such that $t = IS(s) \cdot e \otimes \alpha, \alpha \in \rho(d_k(s \cdot e))$. The table is called $dispute - free$ iff all the disputed rows $s \in S$ are *treated*.

For any $s_1, s_2 \in S \cup R$, s_1 and s_2 are comparable with the help of the following definitions.

- s_1 and s_2 are *compatible*, denoted by $s_1 \equiv s_2$, iff $\forall e \in E, \forall(\alpha_1, \varpi_1 \otimes \beta_1) \in T(s_1, e)$, $\forall(\alpha_2, \varpi_2 \otimes \beta_2) \in T(s_2, e)$, if $suff(\alpha_1)^{|e|} = suff(\alpha_2)^{|e|}$, then $\varpi_1 \otimes$

$\beta_1 = \varpi_2 \otimes \beta_2$. This means that common input parameters produce the same output parameters.

- s_1 and s_2 are *balanced*, denoted by $s_1 \leftrightarrow s_2$, iff $\forall e \in E, \forall(\alpha_1, \varpi_1 \otimes \beta_1) \in T(s_1, e)$, $\exists(\alpha_2, \varpi_2 \otimes \beta_2) \in T(s_2, e)$ such that $\mathit{suff}(\alpha_1)^{|e|} = \mathit{suff}(\alpha_2)^{|e|}$ and $\forall(\alpha_2, \varpi_2 \otimes \beta_2) \in T(s_2, e)$, $\exists(\alpha_1, \varpi_1 \otimes \beta_1) \in T(s_1, e)$ such that $\mathit{suff}(\alpha_2)^{|e|} = \mathit{suff}(\alpha_1)^{|e|}$. This means any input parameter combination on one has been tested on the other.
- s_1 and s_2 are *equivalent*, denoted by $s_1 \cong s_2$, iff $s_1 \leftrightarrow s_2$ and $s_1 \equiv s_2$, i.e., s_1 and s_2 are *balanced* and they remain *compatible*.

A table is *balanced* iff for every $s, t \in S \cup R$ such that $s \equiv t$, $s \leftrightarrow t$. The table is called *closed* iff for each $t \in R$, there exists $s \in S$ such that $s \cong t$. A *closed* table makes sure that no row in R is different from the rows in S that gives out the potential states of the conjecture.

3.2 Algorithm

The algorithm starts by initializing (S, R, E, T) with $E = I$ and $S = R = \emptyset$, i.e., each input symbol makes one column and there are no rows initially. The first step is to add ϵ to S, where ϵ is an empty string. Thus, R will be extended by adding $\epsilon \cdot i$, for all $i \in I$. Table 1 shows the extensions of S, R and E, where $I = \{a, u\}$.

The test cases are constructed by the elements of $(S \cup R) \times E$. Since $S \cup R$ contains the input symbol sequences as well as the parameterized input sequences, the test cases in each case are constructed in the following way:

i) if $s = \omega \in S \cup R$ an input symbol sequence and $e \in E$, then a test case is constructed as $\omega \cdot e \otimes \alpha_1 \cdot \alpha_2$, where α_1 and α_2 are selected from ${D_I}^*$ such that $|\omega| = |\alpha_1|$ and $|e| = |\alpha_2|$.

ii) If $s = (\omega \otimes \alpha_1) \in S \cup R$ a parameterized input sequence and $e \in E$, then a test case is constructed as $\omega \cdot e \otimes \alpha_1 \cdot \alpha_2$, where α_2 will be selected from ${D_I}^+$ such that $|e| = |\alpha_2|$.

The result of each test case is organized in the table by just filling the cells with output sequences, and does not lead to the extension of rows or columns. Let $\omega \otimes \alpha$ be a test case, where $\omega \in I^+, \alpha \in {D_I}^+$, generating a parameterized output sequence $\lambda(q_0, \omega, \alpha) = \varpi \otimes \beta, \varpi \in O^+, \beta \in {D_O}^+$, then the table will be filled as follows:

i) if there exists $s = \omega_1 \in S \cup R, e = \omega_2 \in E$ such that $\omega_1 \cdot \omega_2$ is a prefix of ω or

ii) if there exists $s = \omega_1 \otimes \alpha_1 \in S \cup R$ and $e = \omega_2 \in E$ such that $\omega_1 \cdot \omega_2$ is a prefix of ω and α_1 is a prefix of α,

then there is a prefix $\alpha_p = \mathit{pref}(\alpha)^{|\omega_1 \cdot \omega_2|}$, $\beta_p = \mathit{pref}(\beta)^{|\omega_1 \cdot \omega_2|}$, $\varpi_p = \mathit{pref}(\varpi)^{|\omega_1 \cdot \omega_2|}$ and $T(s, e)$ will be appended by $(\alpha_p, \varpi' \otimes \beta')$, where $\varpi' = \mathit{suff}(\varpi_p)^{|\omega_2|}$, $\beta = \mathit{suff}(\beta_p)^{|\omega_2|}$.

The table is made *balanced* after every test case performed. Whenever it is not *balanced*, find $s, t \in S \cup R, e \in E, (\alpha_1, \varpi_1 \otimes \beta_1) \in T(s, e)$ such that $s \equiv t$ and there does not exist $(\alpha_2, \varpi_2 \otimes \beta_2) \in T(t, e)$ where $\mathit{suff}(\alpha_1)^{|e|} = \mathit{suff}(\alpha_2)^{|e|}$,

then construct test case $IS(t) \cdot e \otimes pref(\alpha)^{|\alpha|-|e|} \cdot suff(\alpha_1)^{|e|}$ where α is selected from $\rho(d_k(t,e))$, for any $d_k(t,e) \subseteq T(t,e), 1 \leq k \leq \eta(T(s,e))$.

The table is made $dispute-free$ after balancing. Let $s \in S$ be *disputed* then find $e \in E$ such that $\eta(T(t,e)) > 1$. Then, for every distinguishing subset $d_k(t,e) \subset T(t,e), 1 \leq k \leq \eta(T(t,e))$, add $IS(s) \cdot e \otimes \alpha$ to R where α is selected from $\rho(d_k(t,e))$. Remove the original row $s \cdot e \in S \cup R$ if it exists. Construct additional test cases for the missing elements of the table.

When the table is made *balanced* and $dispute-free$, it is made *closed*. Whenever it is not *closed*, find $t \in R$ such that $s \ncong t, \forall s \in S$ and move t to S and extend R accordingly. Construct additional test cases for the missing elements of the table.

When table is *balanced*, $dispute-free$ and *closed*, a PFSM conjecture M' is made from the table in the following way:

- Each $s \in S$ is a state of the conjecture
- $\epsilon \in S$ is the initial state

  For each $s \in S, i \in I$, there exists $\eta(T(s,i))$ transitions. Thus, each distinguishing subset $d_k(s,i) \subseteq T(s,i), 1 \leq k \leq \eta(T(s,i))$, defines one transition $\{s, s', i, o, p, f\}$, in which $p = \{suff(\alpha)^1, \forall \alpha \in \rho(d_k(s,i))\}$, $f = \sigma(s,i) = PS(d_k(s,i))$ and s', i are determined by $\delta(s,i,x), \lambda(s,i,x)$, $\forall x \in p$, resp., in the following way:
- $\delta(s,i,x) = t \in S | t \cong (IS(s) \cdot i \otimes \alpha) \in S \cup R, \alpha \in (\rho(d_k(s,i))^*$
- $\lambda(s,i,x) = OS(d_k(s,i))$

The termination of the algorithm is guaranteed by the finite space of states and transitions of the black box component modeled as PFSM. The operations which keep the algorithm extending the table are two, i.e., disputed row treatment and making the table closed.

A row is disputed if a row (or state) has more than one outputs (or possible transitions) for the same input symbol but for different set of parameter values. If a state in the actual component has m different transitions for an input symbol and a parameter value for each transition has been tested during the process, then there will be at most m rows added in the table for such state and input symbol.

A table is not closed when a row r in R is not equivalent to any row in S. Then by definition, r will be moved to S and will represent a state of the conjecture. If there are n states in the actual component, then there will be at most $n-1$ moves from R to S, since there is initially one row in S and there cannot be more than n.

As to balancing the table, it is nothing more than recording output sequences in the existing table for those input parameter values that are not recorded previously. The number of test cases required for balancing the table is calculated as follows. Let $m_{r,e}$ is the number of different input parameter values recorded in $T(r,e), r \in S \cup R, e \in E$, and n_e is the number of different input parameter values recorded in $T(s,e), \forall s \in S \cup R$. Then, the number of test cases required for balancing each $T(r,e)$ is $n_e - m_{r,e}$.

3.3 Illustration

We illustrate the learning algorithm of PFSM model on the example given in Figure 1. The summary of the algorithm is given below.

Input: I, D_I
Output: Conjecture M'
begin
  Initialize (S, R, E, T) by $E = I$, $S = \epsilon$, $R = \epsilon \cdot i, \forall i \in I$;
  Construct the test cases from $(S \cup R) \times E$;
  Organize result in the table accordingly ;
  while *table is not balanced* **or** *not dispute* $-$ *free* **or** *not closed* **do**
    Make the table *balanced* such that for every $s, t \in S \cup R | s \equiv t$, $s \leftrightarrow t$;
    Make the table *dispute* $-$ *free* such that for all $s \in S, e \in E$, where $\eta(T(s, e)) > 1$, s is treated ;
    Make the table *closed* such that for every $t \in R$, there exists $s \in S$ such that $s \cong t$;
  end
  Make a conjecture M' from the table.
end

Algorithm 1: Summary of the Learning Algorithm

We start by initializing (S, R, E, T) with the input symbols from $I = \{a, u\}$ and construct test cases to fill the table, shown in Table 1. In the test cases, we associate parameter values 1 and 5 and balance the table accordingly. Thus, the row ϵ becomes disputed, since $\eta(T(\epsilon, u)) > 1$.

	a	u
ϵ	$(\perp, s \otimes \perp)$	$(1, s \otimes \perp)$, $(5, t \otimes 5)$
a	$((\perp, \perp), s \otimes \perp)$	$((\perp, 1), s \otimes \perp)$, $((\perp, 5), t \otimes 25)$
$u \otimes 1$	$((1, \perp), s \otimes \perp)$	$((1, 1), s \otimes \perp)$, $((1, 5), t \otimes 25)$
$u \otimes 5$	$((5, \perp), s \otimes \perp)$	$((5, 1), s \otimes \perp)$, $((5, 5), t \otimes 5)$

Table 2. Table is not *closed*

	a	u
ϵ	$(\perp, s \otimes \perp)$	$(1, s \otimes \perp)$, $(5, t \otimes 5)$
a	$((\perp, \perp), s \otimes \perp)$	$((\perp, 1), s \otimes \perp)$, $((\perp, 5), t \otimes 25)$
$u \otimes 1$	$((1, \perp), s \otimes \perp)$	$((1, 1), s \otimes \perp)$, $((1, 5), t \otimes 25)$
$u \otimes 5$	$((5, \perp), s \otimes \perp)$	$((5, 1), s \otimes \perp)$, $((5, 5), t \otimes 5)$
aa	$((\perp, \perp, \perp), s \otimes \perp)$	$((\perp, \perp, 1), s \otimes \perp)$, $((\perp, \perp, 5), t \otimes 25)$
au	$((\perp, 5, \perp), s \otimes \perp)$	$((\perp, 5, 1), s \otimes \perp)$, $((\perp, 1, 5), t \otimes 25)$

Table 3. Table is not *dispute* $-$ *free*

We add two parameterized sequences $u \otimes 1$ and $u \otimes 5$ to R and refill the table by constructing test cases for new rows and balance it respectively, shown in Table 2. The table is not closed, since row a in R is not equivalent to any row in

	a	u
ϵ	$(\perp, s \otimes \perp)$	$(1, s \otimes \perp)$, $(5, t \otimes 5)$
a	$((\perp, \perp), s \otimes \perp)$	$((\perp, 1), s \otimes \perp)$, $((\perp, 5), t \otimes 25)$
$u \otimes 1$	$((1, \perp), s \otimes \perp)$	$((1, 1), s \otimes \perp)$, $((1, 5), t \otimes 25)$
$u \otimes 5$	$((5, \perp), s \otimes \perp)$	$((5, 1), s \otimes \perp)$, $((5, 5), t \otimes 5)$
aa	$((\perp, \perp, \perp), s \otimes \perp)$	$((\perp, \perp, 1), s \otimes \perp)$, $((\perp, \perp, 5), t \otimes 25)$
$(a, u) \otimes (\perp, 1)$	$((\perp, 1, \perp), s \otimes \perp)$	$((\perp, 1, 1), s \otimes \perp)$, $((\perp, 1, 5), t \otimes 25)$
$(a, u) \otimes (\perp, 5)$	$((\perp, 5, \perp), s \otimes \perp)$	$((\perp, 5, 1), s \otimes \perp)$, $((\perp, 5, 5), t \otimes 5)$

Table 4. Table is *balanced*, $dispute - free$ and *closed*

S (that contains only one row ϵ and $a \ncong \epsilon$). Thus, we move a to S and extend R accordingly, shown in table 3. Balancing the table makes the row a disputed, as $\eta(T(a, u)) > 1$. Hence, we add two more parameterized sequences in R and construct test cases to fill new rows. Table 4 is *balanced*, $dispute - free$ and *closed*. Figure 2 shows the conjecture from the current table. Note that we can use arbitrary input parameter values every time we construct test cases, whereas in the example, only 1 and 5 are used for sake of simplicity. However, using many different parameter values is more likely to reveal interesting information.

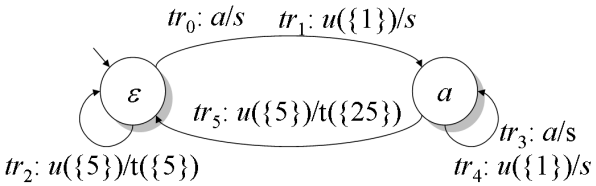

Fig. 2. The first conjecture of the example

3.4 Dealing with Counterexamples

The original learning algorithm for DFA [1], its improvements [14], [7] and its adaptations to more expressive models [9], [2], [10] performs an additional concept on the observation table, i.e., the table must be *consistent* before making the conjecture. The consistency concept can be described informally in the following way. If there are two equivalent rows $s, t \in S$, then all the subsequent rows in $S \cup R$, which extend s, t with some input symbol $i \in I$, must also be equivalent. In other words, the two apparently similar states (i.e., rows in S) must have same successive states for all inputs implied on those states. If the table is found not consistent, then the corresponding input sequence (which makes the successive states different) is added to E. This means that rows are extended with longer input sequences and then new test cases are constructed to fill the table. In this way, two apparently similar states in S become different.

In the learning algorithm of PFSM, we do not perform this concept because any two rows in S remain inequivalent during the whole process. Therefore in-

consistency does not occur in the first iteration of the learning process. If a conjecture made from the table is not correct and there is a counterexample (an input sequence) that rejects the conjecture (the output sequence differs from the conjecture when applying counterexample to the component), then the counterexample is fixed back into the table in order to refine the conjecture, which is considered as the next iteration of the learning process. In all abovementioned algorithms, a counterexample is fixed by adding all its prefixes in S and hence new test cases are constructed for new rows. That is where the inconsistency may occur while adding prefixes in S.

This concept can be avoided altogether if the method of fixing a counterexample in the table is modified in such a way that instead of adding all prefixes in S, we only add the relevant sequence in the table that results in difference between the conjecture and the actual component. Furthermore, this addition will not be reflected in S, so that no two rows in S become equivalent. A general idea is discussed in [3], inspired by [14], applied on DFA algorithm. However, we deal differently in our case which is described below.

Let $c = \omega \otimes \alpha, \omega \in I^+, \alpha \in {D_I}^+$ be a counterexample for the current conjecture. Then c will be fixed in the observation table as follows:

If there exists $s \in S \cup R$ such that $IS(s)$ is the longest prefix of ω then add $e = \mathit{suff}(IS(c))^{|IS(c)|-|IS(s)|}$ in E, if it is not already present. In case where $s = \omega_1 \otimes \alpha_1 \in S \cup R$ a parameterized input sequence and α_1 is not a prefix of α then add $\omega_1 \otimes \mathit{pref}(\alpha)^{|\omega_1|}$ in R. Organize $\lambda(q_0, \omega, \alpha)$ in the table and make it *balanced*, $dispute - free$ and *closed* for a new PFSM conjecture.

We have observed that fixing the counterexample in this way actually gives the same result as fixing the inconsistency in other algorithms. In other algorithms, E is extended only when inconsistency is found, which is reflected after fixing the counterexample. In our explanation, we extend E immediately while fixing the counterexample which keeps the rows in S inequivalent.

4 Integration Testing

In [9], we described the overall testing procedure in which the model is Mealy machine, with adaptations to a restricted form of PFSM in [10]. We suppose that we are provided with a set of components and the architecture of communication linking them. That is, we know for each component its interfaces. Each interface is a set of input and output symbol types and the types of associated parameters. Interface of two components can be pairwise connected, provided they are complementary (inputs and outputs correspond, and parameter types match). In an integrated architecture, non-connected interfaces will be considered as external interfaces to the environment. An example of a composed system of two components M and N is shown in Figure 3.

In order to associate PFSM models to components, we must provide a mapping from interfaces and parameter domains to sets I, O, D_I, D_O for each component. In this mapping, we may omit unrelevant parameters: some expertise may be needed there to identify which parts of the system are of interest. Some high-

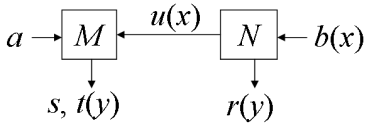

Fig. 3. Example of a Composed System

level description of the integration (e.g. with component diagrams, use cases...) could help in identifying relevant elements. We assume that through this mapping, each machine can be modeled with a PFSM, i.e, all state information will be captured in finite state. We also assume that typical parameter values to be tested are provided for each input: those could be provided by scenarios (esp. for external interfaces) or, failing that, chosen randomly. And for parameters considered not relevant, there should be some mechanism to assign them a value (either a default value, or some value linked to the values of other parameters, e.g. observed values in similar type).

We first learn each component in isolation, using algorithm 1 up to the first conjecture: we call this "unit testing". Thus, we get a PFSM model for each component. Actually, when the conjecture is made, some transitions will be labelled as "unchecked" as will be explained in section 4.2. From that point, we proceed to integration testing, where we connect the actual components using the specified architecture. We also connect the models of components: for this, since PFSM are a restricted form of EFSM, we use the IF tool-set [4] to compute interaction sequences. When we execute a test case, we submit external input symbols along with external input parameter values to the integrated system, observe the external output symbols and the external output parameter values. At the same time, by observing the internal interfaces, we also obtain the input and output sequences of the components. By using the mapping to the inputs and output of the models, and running the corresponding sequences on the integrated model, we can detect any discrepancy between the observed behaviors of components and that of their models. Those discrepancies can then be used as counterexamples to refine the models.

In order to choose integration tests, we can first use some information provided as scenarios or properties of the system, as described below in section 4.1. In any case, we shall be able to use the information from unit testing to derive systematic integration test cases, as described in section 4.2. Additionally, random walk on the model could provide a cheap test generation strategy: it could also be related to a coverage of the "unchecked" transitions.

4.1 Test Generation by Scenario or Model Checking

In component integration, the integrator may have a number of test scenarios for the global interaction of the system with its environment. Additionally, sample parameter values are provided for all external interfaces of the system. For each

test scenario, a test case is constructed, in which the input parameter values are selected according to the ranges specified in the test scenario. In executing the test case, we check two properties:

- Whether the test scenario has been respected. If the test scenario has not been respected, an error has been detected in the system of components.
- Whether the observed behaviors conform to the models of components. If there is a discrepancy between the observed behavior of one component and its model, we go back to the unit testing procedure to refine the model with the input sequence as counterexample.

Another source for test cases could come from property checking. If some properties are specified for the system, then we can model-check those properties on the composed model. Any counterexample for the property could then be run on the system to check whether the actual system also includes a violation of the property. This combination of model-checking with learned models has been quite extensively studied in [5]. If no specific property is provided, we can still check for generic properties. In particular, we could check for livelocks, since our unit testing cannot guarantee that the models do not livelock when integrated (deadlocks are a different matter since we make our models input-enabled).

4.2 Test Generation using Information from Learning Procedure

In the unit testing procedure, in the step of making a conjecture, the set of states is taken from S. When we want to define a transition from a state s for an input symbol and a set of parameter value, we try to identify the corresponding sequence s' in $S \cup R$, through observation recorded in T. If s' is in S, the next state of the transition is that sequence. If s' is in R, we find the sequence $t \in S$ which is equivalent to s'.

In the first case, since the sequences s and s' are all in S, they are not equivalent to each other. So, we are sure that in the real model of the component, the state reached by s and the state reached by s' are different, and there must exist such a transition from the state reached by s to the state reached by s'.

In the latter case, we cannot distinguish the state reached by s' and the state reached by t using the current set E of separating sequences. So, in the conjecture, we assume these two states are the same, and there is a transition from the state reached by s to the state reached by t.

But this conjecture may be wrong. In the real model of the component, the state reached by s' and the state reached by t can be different. These two states can be distinguished by certain sequence. From the point of view of identifying counterexamples for the conjecture, in the integration testing procedure, we should try to separate these states by executing long sequences from them. Based on this observation, we propose the following integration testing technique.

In making a conjecture in the unit testing procedure, for $s \in S$, $i \in I$, $\alpha \in (\rho(d_k(s,i))^*$, if $t = (IS(s) \cdot i \otimes \alpha) \in R$ then we label the transition as *unchecked*, and we record the sequence t with it and refer to it as the *hidden sequence*.

Our test generation strategy for integration testing will be specifically targeted at covering unchecked transitions. For each unchecked transition, we extend its hidden sequence with several parameterized input sequences whose lengths are limited by a predefined threshold k to obtain a group of sequences. From all these sequences obtained, we remove those sequences which have been executed in unit testing, and those sequences in which there is not any interaction with another component. Those sequences are local to a given component, and should be extended to a global test sequence. Therefore, we take rest of the sequences as test purposes, and obtain a group of test cases which contain external inputs/outputs only using the method described in [8]: basically we search the composed model for global sequences whose projections on the local component match the test purpose. Actually, in a single search, we may compute the test cases for several components. By executing these test cases, we may identify counterexamples.

In the example of a composed system shown in Figure 3, component M has $I_M = \{a, u\}$ and $O_M = \{s, t\}$, and component N has $I_N = \{b\}$ and $O_N = \{u, r\}$, respectively. The PFSM model of component M is shown in Figure 1. After unit testing, the first conjecture $M^{(1)}$ of component M is learnt, shown in Figure 2. The PFSM model of component N is shown in Figure 4. It is learnt exactly in its unit testing.

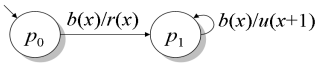

Fig. 4. PFSM Model of Component N

In $M^{(1)}$, among the 6 transitions, transitions tr_1, tr_2, tr_3, tr_4 and tr_5 are unchecked. For unchecked transition tr_2, its hidden sequence is $u(5)$. We extend it to obtain $u(5)\cdot a\cdot a$, $u(5)\cdot a\cdot u(5)$, $u(5)\cdot u(5)\cdot a$, and $u(5)\cdot u(5)\cdot u(5)$. Among them, using $u(5) \cdot a \cdot u(5)$ as test purpose, we obtain a test case $b(4)/r(4) \cdot b(4)/t(5) \cdot a/s \cdot b(4)/t(25)$.

In executing this test case, the expected behavior of component M is $u(5)/t(5)\cdot a/s\cdot u(5)/t(25)$, and the observed behavior is $u(5)/t(5)\cdot a/s\cdot u(5)/t(5)$. This means that a counterexample $u(5) \cdot a \cdot u(5)$ is identified.

Going back to unit testing, we fix the counterexample in the observation table by adding $a \cdot u$ in E and then making the table *balanced*, $dispute - free$ and *closed*, we obtain a new conjecture $M^{(2)}$ for component M. The table is shown in Table 5 and Figure 5 is the conjecture. The new conjecture then will again be put under integration testing with component N and new global test sequences will be generated according to the process described above. This may identify new counterexamples or end the integration process if no discrepancy is found [9]. In the former case, the conjecture will then be refined again through fixing new counterexamples in the table.

	a	u	au
ϵ	$(\bot, s \otimes \bot)$	$(\bot, s \otimes \bot)$, $(5, t \otimes 5)$	$((\bot, 5), (s, t) \otimes (\bot, 25))$, $((\bot, 1), (s, s) \otimes (\bot, \bot))$
a	$((\bot, \bot), s \otimes \bot)$	$((\bot, 1), s \otimes \bot)$, $((\bot, 5), t \otimes 25)$	$((\bot, \bot, 5), (s, t) \otimes (\bot, 25))$, $((\bot, \bot, 1), (s, s) \otimes (\bot, \bot))$
$u \otimes 5$	$((5, \bot), s \otimes \bot)$	$((5, 1), s \otimes \bot)$, $((5, 5), t \otimes 5)$	$((5, \bot, 5), (s, t) \otimes (\bot, 5))$, $((5, \bot, 1), (s, s) \otimes (\bot, \bot))$
$u \otimes 1$	$((1, \bot), s \otimes \bot)$	$((1, 1), s \otimes \bot)$, $((1, 5), t \otimes 25)$	$((1, \bot, 5), (s, t) \otimes (\bot, 25))$, $((1, \bot, 1), (s, s) \otimes (\bot, \bot))$
aa	$((\bot, \bot, \bot), s \otimes \bot)$	$((\bot, \bot, 1), s \otimes \bot)$, $((\bot, \bot, 5), t \otimes 25)$	$((\bot, \bot, \bot, 5), (s, t) \otimes (\bot, 25))$, $((\bot, \bot, \bot, 1), (s, s) \otimes (\bot, \bot))$
$(a, u) \otimes (\bot, 1)$	$((\bot, 1, \bot), s \otimes \bot)$	$((\bot, 1, 1), s \otimes \bot)$, $((\bot, 1, 5), t \otimes 25)$	$((\bot, 1, \bot, 5), (s, t) \otimes (\bot, 25))$, $((\bot, 1, \bot, 1), (s, s) \otimes (\bot, \bot))$
$(a, u) \otimes (\bot, 5)$	$((\bot, 5, \bot), s \otimes \bot)$	$((\bot, 5, 1), s \otimes \bot)$, $((\bot, 5, 5), t \otimes 5)$	$((\bot, 5, \bot, 5), (s, t) \otimes (\bot, 5))$, $((\bot, 5, \bot, 1), (s, s) \otimes (\bot, \bot))$
$(u, u) \otimes (5, 1)$	$((5, 1, \bot), s \otimes \bot)$	$((5, 1, 1), s \otimes \bot)$, $((5, 1, 5), t \otimes 5)$	$((5, 1, \bot, 5), (s, t) \otimes (\bot, 5))$, $((5, 1, \bot, 1), (s, s) \otimes (\bot, \bot))$
$(u, u) \otimes (5, 5)$	$((5, 5, \bot), s \otimes \bot)$	$((5, 5, 1), s \otimes \bot)$, $((5, 5, 5), t \otimes 5)$	$((5, 5, \bot, 5), (s, t) \otimes (\bot, 5))$, $((5, 5, \bot, 1), (s, s) \otimes (\bot, \bot))$
$(u, a) \otimes (5, \bot)$	$((5, \bot, \bot), s \otimes \bot)$	$((5, \bot, 1), s \otimes \bot)$, $((5, \bot, 5), t \otimes 5)$	$((5, \bot, \bot, 5), (s, t) \otimes (\bot, 5))$, $((5, \bot, \bot, 1), (s, s) \otimes (\bot, \bot))$
$(a, a, u) \otimes (\bot, \bot, 5)$	$((\bot, \bot, 5, \bot), s \otimes \bot)$	$((\bot, \bot, 5, 1), s \otimes \bot)$, $((\bot, \bot, 5, 5), t \otimes 5)$	$((\bot, \bot, 5, \bot, 5), (s, t) \otimes (\bot, 5))$, $((\bot, \bot, 5, \bot, 1), (s, s) \otimes (\bot, \bot))$
$(a, a, u) \otimes (\bot, \bot, 1)$	$((\bot, \bot, 1, \bot), s \otimes \bot)$	$((\bot, \bot, 1, 1), s \otimes \bot)$, $((\bot, \bot, 1, 5), t \otimes 25)$	$((\bot, \bot, 1, \bot, 5), (s, t) \otimes (\bot, 25))$, $((\bot, \bot, 1, \bot, 1), (s, s) \otimes (\bot, \bot))$
$(u, a, u) \otimes (5, \bot, 1)$	$((5, \bot, 1, \bot), s \otimes \bot)$	$((5, \bot, 1, 1), s \otimes \bot)$, $((5, \bot, 1, 5), t \otimes 5)$	$((5, \bot, 1, \bot, 5), (s, t) \otimes (\bot, 5))$, $((5, \bot, 1, \bot, 1), (s, s) \otimes (\bot, \bot))$
$(u, a, u) \otimes (5, \bot, 5)$	$((5, \bot, 5, \bot), s \otimes \bot)$	$((5, \bot, 5, 1), s \otimes \bot)$, $((5, \bot, 5, 5), t \otimes 5)$	$((5, \bot, 5, \bot, 5), (s, t) \otimes (\bot, 5))$, $((5, \bot, 5, \bot, 1), (s, s) \otimes (\bot, \bot))$

Table 5. Table is *balanced*, $dispute - free$ and *closed* after fixing the counterexample

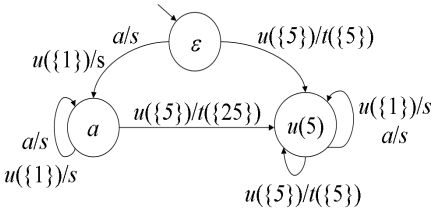

Fig. 5. Conjecture $M^{(2)}$ of Component M

5 Conclusion

We have presented an approach that makes it possible to use model-based testing techniques, in particular test generation for integration testing, in the absence of initial models. We extend previous work done in this direction [7], [9], [5] to deal with arbitrary data values, avoiding the complexity of expanding into DFA or FSM models. The model is richer than the models used by [10] or [2]. We use an incremental testing approach where new interoperability tests can be derived to check systematically the models derived from previous observations. From those tests, refined models of the system can be built, or faults in the system can be identified, as explained in [9].

We are currently working on a tool, called RALT (Rich Automata Learning and Testing), to run the approach on case studies to be provided by France Telecom. We have already implemented the learning algorithms for DFA [1], for Mealy machine [9] and for simple parameterized machine [10], and need to interface to actual test drivers, so that we can compare them all.

We also consider research perspectives to deal with even more complex models. In particular, we could try to move closer to EFSM models by incorporating variables. To circumvent the hidden nature of state structure in black boxes, we could either rely on additional structure information provided by the integrator (moving from black to some kind of grey box) or use some heuristics to differentiate control states from variables. Other direction is to consider sufficient information (e.g., parts of source code) and derive complex models, as performed

in [6], [15]. We are also investigating other types of test generation strategies for integration testing.

References

1. Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2:87–106, 1987.
2. Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In *FASE*, volume 3922 of *LNCS*, pages 107–121. Springer, 2006.
3. Therese Berg and Harald Raffelt. Model checking. In *Model-Based Testing of Reactive Systems*, pages 557–603, 2004.
4. Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF toolset. In *SFM*, volume 3185 of *LNCS*, pages 237–267. Springer, June 2004.
5. Edith Elkind, Blaise Genest, Doron Peled, and Hongyang Qu. Grey-box checking. In *FORTE*, volume 4229 of *LNCS*, pages 420–435. Springer, 2006.
6. Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.
7. Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-specific optimization in automata learning. In *CAV*, volume 2725 of *LNCS*, pages 315–327. Springer, 2003.
8. Ousmane Koné and Richard Castanet. Test generation for interworking systems. *Computer Communications*, 23(7):642–652, 2000.
9. Keqin Li, Roland Groz, and Muzammil Shahbaz. Integration testing of components guided by incremental state machine learning. In *TAIC PART*, pages 59–70. IEEE Computer Society, 2006.
10. Keqin Li, Roland Groz, and Muzammil Shahbaz. Integration testing of distributed components based on learning parameterized i/o models. In *FORTE*, volume 4229 of *LNCS*, pages 436–450. Springer, 2006.
11. D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In *FORTE*, volume 156 of *IFIP Conference Proceedings*, pages 225–240. Kluwer, 1999.
12. Alexandre Petrenko, Sergiy Boroday, and Roland Groz. Confirming configurations in EFSM testing. *IEEE Trans. Softw. Eng.*, 30(1):29–42, 2004.
13. T. Ramalingom, Krishnaiyan Thulasiraman, and Anindya Das. Context independent unique state identification sequences for testing communication protocols modelled as extended finite state machines. *Computer Communications*, 26(14):1622–1633, 2003.
14. Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. In *Machine Learning: From Theory to Applications*, pages 51–73, 1993.
15. Neil Walkinshaw, Kirill Bogdanov, and Mike Holcombe. Identifying state transitions and their functions in source code. In *TAIC PART*, pages 49–58. IEEE Computer Society, 2006.