Integration Testing of Distributed Components Based on Learning Parameterized I/O Models

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

[1] LSR - IMAG
BP 72, F-38402 St Martin D'Hères Cedex, France
{Keqin.Li, Roland.Groz}@imag.fr
[2] France Telecom R&D
BP 98, 38243 Meylan Cedex, France
muhammad.muzammilshahbaz@orange-ft.com

Abstract. The design of complex systems, e.g., telecom services, is usually based on the integration of components (COTS). When components come from third party sources, their internal structure is usually unknown and the documentation is scant or inadequate.

Our work addresses the issue of providing a sound support to component integration in the absence of formal models. We consider components as black boxes and use an incremental learning approach to infer partial models. At the same time, we are focusing on the richer models that are more expressive in the designing of complex systems. Therefore, we propose an I/O parameterized model and an algorithm to infer it from a black box component. This is combined with interoperability testing covering models of the components.

1 Introduction

The design of new software systems, such as telecom services, is more and more based on the integration of components from third party sources (COTS), loosely coupled in a distributed architecture (e.g., web services). Testing the behavior of the assembly is important to build confidence in the system. In order to base testing on a sound and systematic basis, it has become common to use models. In reality, COTS are rarely provided with formal descriptions.

A general approach [4,2,3] is to generate formal models of COTS through their incremental learning. In [6], we proposed an approach to learn I/O models of COTS (using a slight modification of Angluin's Algorithm [1]), and define an Integration Testing Procedure based upon these models. Our current work addresses the issue of learning richer models that are more expressive in the design of complex systems. The goal is to help the integrator in deriving systematic tests to check component interactions. It is well known that typical interoperability problems are often related to incompatibility of data values that did not appear when components are tested in isolation, but are revealed by feeding the outputs of one component as inputs to another component. This is why we concentrate on a model where parameter values are taken into account. From those models, we compute systematic cross component interactions with classical test generation techniques for collections of automata.

1.1 Assumptions

Our basic assumptions are as follows.

- The components we deal with are viewed as black boxes. Only their interfaces are known, which means that we know at least their input types, although the actual parameter values may depend on the behaviors exercised.
- Although no global specification of the system is available, the integrator has a number of test scenarios for the global interaction of the system with its environment. Additionally, sample parameter values are provided for all interfaces of components.
- All internal and external interfaces can be observed in integration testing, but only external (non-integrated) interfaces are controllable, i.e. we can send input sequences through these interfaces to test the components. We assume our test harness makes it possible to observe connected interfaces.
- Inputs from the environment will only be provided on stable states of the system, viz. when all components are waiting for external stimuli and will not make any internal move. This corresponds to the notion of slow environment in system verification or quiescence in testing theory. We also assume that each input to a component triggers at most one output. Interaction between components is asynchronous, and at any time at most one "message in transit" holds in the system.
- We shall not attempt to derive a complete model of the components. COTS offer many functions. We shall just derive sub-models that correspond to the behaviors exercised in the integrated system. Even for that restricted part of the components, the models derived will be approximations in line with the testing goals (i.e., the level of confidence required).

1.2 Our Approach

In the absence of formal models for components, model inference from observations is a key point. Angluin [1] has proposed an algorithm that infers a Deterministic Finite Automaton (DFA) from observations of component's behavior. In [6], we proposed an extension of Angluin's algorithm that works with I/O automata. In this paper, we are dealing with a richer model that contains inputs and outputs along with the parameter values. These models are more applicable when the input set is typically very large. Then we can distinguish input types and their possible parameter values and model into a compact finite state machine, which we call PFSM (Parameterized Finite State Machines). Since existing algorithm for DFA inference uses number of queries, which grow polynomially with the size of alphabet, they are not well-suited for this situation. If some parameters are irrelevant or never used, the algorithm should not be disturbed by their presence. Certain adaptations of this algorithm have been tested for parameterized FSM e.g. in [2] but it does not cater for output parameters in the model.

Our approach is to further modify the above algorithms to conjecture a PFSM model of a component and also find a practical source for getting counterexamples when the conjecture is wrong or not suitable for integration. In requirements engineering approaches [7,8], the equivalence query used to get counterexamples is provided by an expert. Here we follow the approach used in [4,3] where the query is implemented by testing the integrated system which acts as an oracle. The outline of the integration methodology is as follows.

1. In the first step, an input alphabet is defined for each component C. This corresponds to the invocations on external interfaces with all the parameter values that are considered relevant (starting with those from scenarios or use cases defined for the system or provided for internal interfaces).
2. Each component is (unit) tested separately using the learning algorithm until balanced, closed and consistent observation tables for it are found. The output alphabet is determined along with output parameters. This provides the first model $C^{(1)}$ for C.
3. The components are integrated. This means that some of the outputs of one component will appear as inputs on the connected interface of another component. The assembly is tested in two stages.
4. In a first stage, we systematically test the provided system-wide scenarios expected from the assembly. In that stage, scenarios act as oracles. Faults can be detected, or a discrepancy with the inferred model may be identified, leading to incremental refinement of the model.
5. In a second stage, we generate (interoperability) tests from the models of the components. The actual outputs observed (both internal and to the environment) are recorded and compared to those provided by the models. Tests are performed until a discrepancy between predictions from the model is found or some coverage criteria on the model is achieved. Classifying discrepancies as faults may require expert input.
6. In both stages, discrepancies can lead to model refinement. The counterexamples found are injected in the learning algorithm to extend the models and the stage is iterated with the new $C^{(i+1)}$ components.

The rest of the paper is organized as follows. Section 2 presents unit testing of components and Section 3 presents their integration testing. An example is given in the Section 4. Finally, Section 5 concludes the paper.

2 Unit Testing

The first stage in our approach is unit testing, in which the components are tested individually. For each component C, its inputs are modelled as a set of input symbols $I_C^{(1)}$. This may be provided by the designer of the component, or abstracted from the informal descriptions or some preliminary testing of the component by the tester. With $I_C^{(1)}$, the tester performs unit testing using our learning algorithm and builds an initial model $C^{(1)}$. The model is an extension of FSM which incorporates inputs and outputs along with the parameter values. We call this model as Parameterized Finite State Machine (PFSM). At the same time, this extension can be considered as a restricted form of Extended Finite State Machine (EFSM), in the sense that all the context information are stored in states without the help of variables, and the knowledge of state and input can determine the transition. The next section describes PFSM formally and then its inferring algorithm is presented.

2.1 PFSM Model

A *Parameterized Finite State Machine* (PFSM) $M = \{Q, I, O, D_x, D_y, \delta, \lambda, \sigma, q_0\}$, where

- Q is finite non-empty set of states,
- I is finite set of input symbols,
- O is finite set of output symbols,
- $q_0 \in Q$ is the initial state,
- D_x is a set of possible values of input parameters,
- D_y is a set of possible values of output parameters,
- $\delta : Q \times I \longrightarrow Q$ is a next state function,
- $\lambda : Q \times I \longrightarrow O$ is an output function,
- $\sigma : Q \times I \longrightarrow {D_y}^{D_x}$ is an output parameter function. ${D_y}^{D_x}$ is the set of all functions from D_x to D_y.

When a PFSM model M is in the state $q \in Q$ and receives an input $i \in I$ along with the parameter value $x \in D_x$, M moves to the next state given by $q' = \delta(q, i) \in Q$, and produces an output given by $o = \lambda(q, i) \in O$, along with the output parameter value given by $f(x) = \sigma(q, i)(x)$.

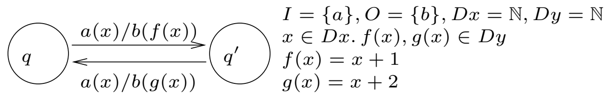
In order to elaborate a complete parameterized output function for a state $q \in Q$, $i \in I$ and $x \in D_x$, we write $\lambda(q, i(x)) = o(f(x))$, where $o \in O$ is determined by $\lambda(q, i)$ and $f(x)$ is determined by $\sigma(q, i)(x)$.

Then, we extend the functions from input symbols to sequences as usual: for a state q_1, an input sequence $\alpha = i_1(x_1), ..., i_k(x_k)$ takes M successively to the states $q_{j+1} = \delta(q_j, i_j), j = 1, ..., k$, with the final state $\delta(q_1, \alpha) = q_{k+1}$, and produces an output sequence $\lambda(q_1, \alpha) = o_1(f_1(x_1)), ..., o_k(f_k(x_k))$, where each $o_j(f_j(x_j)) = \lambda(q_j, i_j(x_j)), j = 1, ..., k$.

In the definition of PFSM M, in case D_x and D_y are obvious from context or trial, they can be omitted.

For input string $\alpha_i = i_1, ..., i_k (i_j \in I, 1 \leq j \leq k)$, and input parameter string $\alpha_p = x_1, ..., x_k (x_j \in D_x, 1 \leq j \leq k)$, we define their association as $\alpha_i \otimes \alpha_p = i_1(x_1), ..., i_k(x_k)$. The association of output string and output parameter string can be defined similarly.

We only consider *input enabled* PFSM, that is when $dom(\delta) = dom(\lambda) = Q \times I$. A machine can be made input enabled by adding loop back transitions on a state for all those inputs which are not acceptable for that state. Such transitions contain a special symbol Ω in place of an output from O. There may be some transitions in a PFSM model which contain no parameters, i.e., no input parameter value or output parameter value is associated with the respective inputs or outputs on the transitions. An example of PFSM model is given in the Figure 1.



q
q'
$a(x)/b(f(x))$
$a(x)/b(g(x))$
$I = \{a\}, O = \{b\}, Dx = \mathbb{N}, Dy = \mathbb{N}$
$x \in Dx.\ f(x), g(x) \in Dy$
$f(x) = x + 1$
$g(x) = x + 2$

Fig. 1. An example of PFSM model

2.2 Observation Tables

We assume that the reader is familiar with the original Angluin algorithm [1]. Here we explain our modifications with respect to PFSM inference.

In this work, an unknown PFSM $M = (Q, I, O, D_x, D_y, \delta, \lambda, \sigma, q_0)$ with known input symbols I is used to model a component C. Since we can submit any input sequence with parameters to the component and observe the corresponding output sequence with parameters, for any input sequence $\alpha = i_1(x_1), ..., i_k(x_k)(i_j \in I, 1 \leq j \leq k)$, $\lambda(q_0, \alpha)$ is known. We also assume that each component can be reset to its initial state before each test.

Basic Structure of Observation Tables. In the testing procedure, the observed behavior of the component C is recorded into two Observation Tables *Primary Table* (PT) and *Auxiliary Table* (AT), denoted by (S, E, T) and (S, E, T') respectively. The original Angluin's observation table is reflected in the Primary Table after being adapted in the way proposed in [6], the Auxiliary Table will record information on parameters. S is a nonempty finite prefix-closed set of input strings (representing potential states of the PFSM). E is a nonempty finite suffix-closed set of input strings (separating potential states of the PFSM), but the suffix ε does not belong to E. T is a finite function mapping $((S \cup S \cdot I) \times E)$ to O^*. T' is a finite function mapping $((S \cup S \cdot I) \times E)$ to $2^{\{(\alpha_p, \beta_p) | \alpha_p \in {D_x}^+, \beta_p \in {D_y}^+\}}$.

In PT, for each $s \in (S \cup S \cdot I)$ and $e \in E$, $T(s, e) = t$, $t \in O^*$, such that $|t| = |e|$, and $\lambda(q_0, s \cdot e) = \lambda(q_0, s) \cdot t$.

In AT, for each $s \in (S \cup S \cdot I)$ and $e \in E$, if $(\alpha_p, \beta_p) \in T'(s, e), \alpha_p \in {D_x}^+, \beta_p \in {D_y}^+$, then $|\alpha_p| = |\beta_p| = |e|$, and $\forall \gamma_p \in {D_x}^{|s|}$, $\lambda(q_0, (s \cdot e) \otimes (\gamma_p \cdot \alpha_p)) = \lambda(q_0, s \otimes \gamma_p) \cdot (T(s, e) \otimes \beta_p)$.

Initially $S = \{\varepsilon\}$ and $E = I$. These sets are updated during the testing procedure.

Each table can be visualized as a two-dimensional array with rows labelled by the elements of $S \cup S \cdot I$ and columns labelled by the elements of E, with the entry for row s and column e equal to $T(s, e)$ and $T'(s, e)$ respectively. For $s \in S \cup S \cdot I$, $row_{PT}(s)$ denotes the finite function f in PT from E to O^+ defined by $f(e) = T(s \cdot e)$, $row_{AT}(s)$ denotes the finite function f' in AT from E to $2^{\{(\alpha_p, \beta_p) | \alpha_p \in Dx^+, \beta_p \in Dy^+\}}$ defined by $f'(e) = T'(s \cdot e)$.

Properties of Observation Tables. In the original Angluin's algorithm, the strings in S represent candidate states of the automaton being learned. The rows in observation table are compared to differentiate the states in the conjecture. In this work, we follow the principle of Angluin's algorithm and adapt it.

In order to differentiate states in the conjecture, in addition to comparing rows in PT, we need to compare rows in AT. Since $T'(s, e)$ is a set of parameter string pairs, *compatibility*, rather than *equality*, is used in comparing rows in AT. Let $s_1, s_2 \in S \cup S \cdot I$ and $e \in E$, we say two sets $T'(s_1 \cdot e)$ and $T'(s_2 \cdot e)$ are *compatible*, denoted as $T'(s_1 \cdot e) \equiv T'(s_2 \cdot e)$ iff $\forall (\alpha_p, \beta_p) \in T'(s_1 \cdot e), \forall (\alpha'_p, \beta'_p) \in T'(s_2 \cdot e)$, if $\alpha_p = \alpha'_p$ then $\beta_p = \beta'_p$. Two rows in AT are *compatible*, i.e., $row_{AT}(s_1) \equiv row_{AT}(s_2)$ iff $T'(s_1 \cdot e) \equiv T'(s_2 \cdot e), \forall e \in E$. We write $T'(s_1 \cdot e) \not\equiv T'(s_2 \cdot e)$ and $row_{AT}(s_1) \not\equiv row_{AT}(s_2)$ as sets and rows are *incompatible*. For example, $\{(1, 2), (2, 3)\} \equiv \{(5, 6), (2, 3)\}$, but $\{(1, 2), (2, 3)\} \not\equiv \{(2, 4), (3, 5)\}$.

we define $s_1 \cong s_2$, iff $row_{PT}(s_1) = row_{PT}(s_2) \wedge row_{AT}(s_1) \equiv row_{AT}(s_2)$.

Let us consider the following example: $s_1, s_2, s_3 \in S \cup S \cdot I$, $E = \{e\}$, and $T(s_1, e) = T(s_2, e) = T(s_3, e)$. $T'(s_1, e) = \{(1, 2), (2, 3)\}$, $T'(s_2, e) = \{(2, 4), (3, 5)\}$, and $T'(s_3, e) = \{(5, 6)\}$. So, we have $s_1 \cong s_3$, $s_2 \cong s_3$, but $s_1 \ncong s_2$. In this case, when deriving states from strings in S, we know s_1 and s_2 correspond to different states, e.g., q_1 and q_2, but we do not know which state s_3 corresponds to. This is because in $T'(s_3, e)$ there is not any element in the form of $(2, y)$.

Based on this observation, we introduce the concept of *balanced* observation tables. The observations tables are called *balanced* provided that $\forall s_1, s_2, s_3 \in S \cup S \cdot I$ and $e \in E$, such that $T(s_1, e) = T(s_2, e) = T(s_3, e)$, if $\exists \alpha_p \in {D_x}^+$, $\beta_{p1}, \beta_{p2} \in {D_y}^+$, s.t., $(\alpha_p, \beta_{p1}) \in T'(s_1, e)$, $(\alpha_p, \beta_{p2}) \in T'(s_2, e)$, and $\beta_{p1} \neq \beta_{p2}$, then $\exists \beta_{p3} \in {D_y}^+$, s.t., $(\alpha_p, \beta_{p3}) \in T'(s_3, e)$. In this previous example, if $T'(s_3, e) = \{(5, 6), (2, 3)\}$, the observation tables are balanced, and $s_1 \cong s_3$, $s_2 \ncong s_3$, and $s_1 \ncong s_2$.

Now, we have the following lemma:

Lemma 1. *In balanced observation tables,* $\cong$ *is an equivalence relationship.*

The proof of the lemma can be found in [5].

For $s_1, s_2 \in S \cup S \cdot I$, if $s_1 \cong s_2$ then s_1 is in the equivalence class of s_2. So we can define $[s], s \in S$ an equivalence class of rows where each row is equivalent to s.

Like in original Angluin's algorithm, the observation tables PT and AT are called *closed* if for each t in $S \cdot I$ there exists an s in S such that $t \cong s$. The observation tables PT and AT are called *consistent* if for every $s_1, s_2 \in S$, such that $s_1 \cong s_2$, it holds that $s_1 \cdot i \cong s_2 \cdot i$, for all $i \in I$.

Making Conjecture from Observation Tables. When the observation tables (S, E, T) and (S, E, T') are balanced, closed and consistent, a conjectured PFSM $M(S, E, T, T') = (Q, I, O, \delta, \lambda, \sigma, q_0)$ can be made from the tables as follows:

- $Q = \{[s] | s \in S\}$,
- $q_0 = [\varepsilon]$,
- $\delta([s], i) = [s \cdot i]$,

– $\lambda([s], i) = T(s \cdot i)$,
– $\sigma([s], i) = \bigcup\limits_{t \in [s]} T'(t \cdot i)$.

The property of the conjecture is stated in a theorem below. A full proof of the theorem can be seen in the technical report [5].

Theorem 1. *If (S, E, T) and (S, E, T') are balanced, closed and consistent observation tables, then the PFSM $M(S, E, T, T')$ is consistent with the primary table (S, E, T) and auxiliary table (S, E, T'). Any other PFSM consistent with (S, E, T) and (S, E, T') but inequivalent to $M(S, E, T, T')$ must have more states.*

2.3 Unit Testing (Learning) Procedure

The unit testing (learning) procedure is described as follows:

1. Start with $S = \{\varepsilon\}$ and $E = I$. All elements in PT are unknown, and all elements in AT are empty sets.
2. Construct test cases for unknown elements in PT, and record the outputs in PT and AT. For $s = i_1, ..., i_m \in S \cup S \cdot I$ and $e = i_{m+1}...i_{m+n} \in E$, choose input parameter values from D_x to construct input parameter string $\alpha_p = x_1, ..., x_{m+n}(x_j \in D_x, 1 \leq j \leq m+n)$, provide $(s \cdot e) \otimes \alpha_p$ to the component as test case, and obtain the output $o_1(y_1), ..., o_m(y_m), ..., o_{m+n}(y_{m+n})$. Set $T(s, e) = o_{m+1}, ..., o_{m+n}$, and include $(x_{m+1}, ..., x_{m+n}, y_{m+1}, ..., y_{m+n})$ in $T'(s, e)$.
3. Make PT and AT balanced. Whenever they are not balanced, find $s_1, s_2, s_3 \in S \cup S \cdot I$, $e \in E$, $T(s_1 \cdot e) = T(s_2 \cdot e) = T(s_3 \cdot e)$, $\alpha_p \in D_x^+$, $\beta_{p1}, \beta_{p2} \in D_y^+$, such that $(\alpha_p, \beta_{p1}) \in T'(s_1 \cdot e)$, $(\alpha_p, \beta_{p2}) \in T'(s_2 \cdot e)$, $\beta_{p1} \neq \beta_{p2}$, but $\nexists(\alpha_p, \beta_{p3}) \in T'(s_3 \cdot e)$. Construct $(s_3 \cdot e) \otimes (\gamma_p \cdot \alpha_p)$ as test case in which γ_p is any input parameter string of length $|s_3|$. Provide the test case to the component and record the output in PT and AT.
4. Check whether PT and AT are closed. If not, find s_1 in S and i in I such that $s_1 \cdot i \ncong s$ for all $s \in S$. Add the string $s_1 \cdot i$ to S in both tables and go back to step 2 to fill missing elements.
5. Check whether PT and AT are consistent. If not, find s_1 and s_2 in S, e in E, and i in I such that $s_1 \cong s_2$, but $T(s_1 \cdot i \cdot e) \neq T(s_2 \cdot i \cdot e)$ or $T'(s_1 \cdot i \cdot e) \not\equiv T'(s_2 \cdot i \cdot e)$. Add the string $i \cdot e$ to E in both tables and go back to step 2 to fill missing elements.
6. Now, PT and AT are balanced, closed and consistent. Make conjecture PFSM $M = (S, E, T, T')$.

Balanced, closed and consistent observation tables of the example in the Figure 1 are shown in the Figure 2. In the example, $row_{PT}(\varepsilon) = row_{PT}(a) = row_{PT}(aa)$ because of same output symbol in all rows. On the other hand, $row_{AT}(\varepsilon) \not\equiv row_{AT}(a)$ because of $\alpha_p = 2$ that makes $T'(\varepsilon, a) \not\equiv T'(a, a)$. We also have $row_{AT}(aa) \equiv row_{AT}(\varepsilon)$ and $row_{AT}(aa) \not\equiv row_{AT}(a)$.

	PT
	a
ε	b
a	b
aa	b

	AT
	a
ε	(1,2)(2,3)
a	(2,4)(3,5)
aa	(5,6)(2,3)

$$f'(x) = \begin{cases} 2, \, x = 1 \\ 3, \, x = 2 \\ 6, \, x = 5 \end{cases}, \quad g'(x) = \begin{cases} 4, \, x = 2 \\ 5, \, x = 3 \end{cases}$$

Fig. 2. Balanced, closed and consistent observation tables of PFSM example in the Figure 1 are shown (left). The learned output parameter functions f' and g' are also shown (right).

So, $\varepsilon \cong aa \ncong a$, i.e., $[\varepsilon]$ and $[a]$ are two different states. The conjecture from the table corresponds to the PFSM model in the Figure 1. The learned output parameter functions f' and g' are in Figure 2, too.

In integration testing, counterexamples and new input symbols can be identified. The process of dealing with them is similar as described in [6].

3 Integration Testing

At the end of unit testing, a conjecture PFSM is obtained for each component. Then the integration testing procedure begins. In this procedure, the components are integrated, and their joint behaviors are tested. Normally, several components can be integrated. The integration testing procedure of two components is illustrated in the following.

Suppose there are two components M and N. Their internal structures are not known, so they are considered as two black boxes. Initially, their sets of input symbols are known as I_M and I_N, respectively. In the unit testing (learning) of them, the initial models $M^{(1)} = (Q_M, I_M, O_M, \delta_M, \lambda_M, \sigma_M, q_{M0})$ and $N^{(1)} = (Q_N, I_N, O_N, \delta_N, \lambda_N, \sigma_N, q_{N0})$ are constructed.

3.1 Integration Testing Procedure

In [6], the integration testing architecture and procedure are described in which the model is Finite State Machine (FSM). In this paper, we follow the principle of [6] and adapt to the PFSM model.

For PFSM $M = (Q_M, I_M, O_M, \delta_M, \lambda_M, \sigma_M, q_{M0})$, we define the projection operator $\downarrow$, which projects M to an FSM $M\downarrow = (Q_M, I_M, O_M, \delta_M, \lambda_M, q_{M0})$. It can be proved that if M is input deterministic and input enabled, $M\downarrow$ is input deterministic and input enabled, too.

In integration testing, a test case is a sequence of tuples in which external input symbol and parameter value, and the expected external output symbol and parameter value are specified. According to the architecture described in [6], when we execute a test case, the external interfaces can be controlled, and all the interfaces can be observed. Thus in addition to comparing the external output symbols and parameter values with expected ones, we also obtain the input and output sequences with parameter values of the two components respectively.

The integration testing procedure can be divided into two stages.

The first stage is similar to Scenario Testing in [6], in which test cases are constructed according to test scenarios. In this work, since a range of input parameters and constraints on output parameter values are specified in test scenarios, the input parameter values are selected according to the ranges during constructing test cases.

In executing the test case, in addition to checking whether the test scenario has been respected, we check 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.

In order to achieve a certain coverage of the ranges specified in test scenarios, each test scenario can yield several test cases. When all the test cases have successfully been executed, we begin the second stage.

In the second stage, test cases are constructed one by one according to a certain test generation strategy. First, following the Integration Testing procedure for FSMs specified in [6], test cases are generated based on $M^{(1)} \downarrow$ and $N^{(1)} \downarrow$. Then, input parameters are selected according to a certain policy to form a complete test case.

Whenever one test case has been generated, we execute it. We check whether the observed behaviors conform to the models of components, and go back to unit testing procedure if counterexample has been found.

This stage and thus the integration testing procedure terminate when the coverage criterion chosen by the test generation strategy is satisfied.

In both stages of the integration testing procedure, after executing a test case and obtaining the real parameterized output string, there are several possibilities:

- The real parameterized output string is exactly the expected one. In this case, we continue to construct and execute the next test case.
- The real parameterized output string is the expected one except for some transitions the executed input parameter values have not been specified in the models. In this case, we record these input/output parameter value pairs in the corresponding cells in AT, and update the corresponding output parameter functions in the models. Then, we continue to construct and execute the next test case.
- The real output symbols are the expected ones, but there are some output parameters which are different from expected ones. In this case, we have found a parameter counterexample. We record these input/output parameter value pairs in the corresponding cells in AT, go back to unit testing to make the observation tables balanced, closed and consistent, and make another conjecture.
- The real output symbols are different from expected ones. In this case, we have found an I/O counterexample. Or For a certain component, some new input symbols have been produced by other components. For the two cases, we go back to unit testing and follow the process specified in [6].

3.2 The Relationship Between Unit Testing and Integration Testing

In unit testing, some input sequences have been executed on a component. Based on the output sequences observed, the closed and consistent observation table has been constructed, and a conjecture of the PFSM has been made.

In integration testing, from the point of view of the component, e.g. M, more input sequences are checked. There are several possibilities to introduce "new" information into these sequence:

– Symbols produced by the other component. When being integrated, some outputs of component N are given to component M as inputs. And these symbols may have not been included in I_M. So, new input symbols are identified. In our approach, this "mismatch" is identified during integration testing, rather than comparing O_N and I_M directly. Thus, only those symbols which appear in the interaction are considered.
– Parameter values produced by the other component. These values are generated in the integration, and may have not been tried in unit testing.
– Test scenario. In a test scenario, along with the pairs of input and output symbols, the parameter values being interested are provided. And some of the values may have not been used in unit testing.
– The second stage of integration testing. In this stage, more new parameter values are used to uncover the behaviors of the integrated system.

With all these "new" information, more behaviors of the components and the integrated system can be observed in the integration testing procedure.

3.3 Result of Integration Testing

At the end of integration testing, for each component, we have a model, which is consistent with all the tests that have been passed. And as stated by Theorem 1, If the component and the model have the same numbers of states, they are equivalent to each other. At the same time, the joint behavior of these components have been systematically tested. Using the approach described in [6], a transition coverage is achieved. Faults could be discovered during integration test execution.

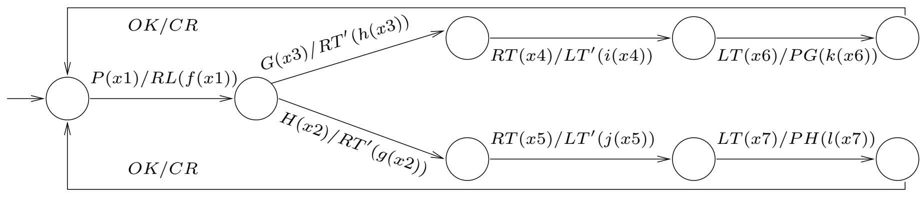
4 Example

We illustrate our component integration strategy using a simple example. Suppose an integrator is developing a travel agency web application, in which two components have been identified, i.e., Room Reservation and Travel Agent.

4.1 Room Reservation Component

The simplified behavior of the component Room Reservation is as follows. The component starts working when a name of the place is given from the external enviornment. It provides a list of residences depending upon the place it is given. The residence can be either a Hotel or a Guest House. Then it takes one residence as input and outputs a list of room types particular for that residence. When one of the room types is given, the component responds with the list of luxury types offered with the room. When one of the luxury type is provided, the components gives out its corresponding price. Finally, it confirms reservation upon an OK signal.

The component can be described as a PFSM model. The inputs, outputs and associated parameter functions are shown in Figure 3. For simplicity, not all the transitions are shown. For each state, if there is no transition for certain input, the machine outputs Ω and stays in the state. Also, D_x and D_y may be infinite but the figure shows some of the possible input parameter values and their corresponding output parameter values. The abbreviations used in the example are also given in the figure.



P: Place, H: Hotel, G: Guest House, RT: Room Type, LT: Luxury Type, OK: OK, RL: Residence List, RT': Room Types, LT': Luxury Types, PH: Price for Hotel, PG: Price for Guest House, CR: Confirm Reservation, PAR: Paris, LDN: London, Htn: Hilton Hotel, Sh: Sherton Hotel, YR: Youth Residence (Guest House), Vil: Villa (Guest House), Sgl: Single, Dbl: Double, Std: Standard, Dlx: Delux, Exe: Executive, Stu: Studio, Dor: Dormitory.

x1: Place Name as PAR, LDN, ...	f(x1): List of Residencies for place x1 as {Htn,Sh,YR,Vil,...}
x2: Hotel Name as Htn, Sh, ...	g(x2): List of Room Types for Hotel x2 as {Sgl,Dbl,...}
x3: Guest House Names as YR, Vil ...	h(x3): List of Room Types for Guest House x3 as {Sgl,Dbl,Dor,...}
x4: Guest House Room Type as Sgl, Dor, ...	i(x4): List of Luxury Types for Guest House Room Type x4 as {Std,Dlx,Stu,...}
x5: Hotel Room Type as Sgl, Dbl, ...	j(x5): List of Luxury Types for Hotel Room Type x5 as {Std,Dlx,Exe,...}
x6: Guest House Luxury Type as Std, Dlx, Stu, ...	k(x6): Cost for Guest House Luxury Type x7 as 50$,60$,...
x7: Hotel Luxury Type as Std, Dlx, Exe, ...	l(x7): Cost for Hotel Luxury Type x7 as 50$,60$,...

Fig. 3. PFSM model of Room Reservation Component

4.2 Unit Testing of Room Reservation Component

In the unit testing procedure, the component Room Reservation is considered as a black box and $I_M^{(1)}$ is known as $\{P, H, G, RT, LT, OK\}$. The *Learner* constructs tables PT and AT for the component. Initially $S = \{\varepsilon\}$, $E = I_M^{(1)}$. The tester execute several test cases with different input parameter values from D_x to fill the tables. Finally when the observation tables are balanced, closed and consistent, a conjecture is made. The Figure 4 shows PT for Room Reservation component. The corresponding AT is shown in the Figure 5. For sake of simplicity, the rows which contain Ω in all columns of the table are omitted. The figure 6 (right) shows the conjecture accompanied by the input and output parameter values used during the unit testing.

	P	H	G	RT	LT	OK
ε	RL	Ω	Ω	Ω	Ω	Ω
P	Ω	RT'	RT'	Ω	Ω	Ω
P-H	Ω	Ω	LT'	Ω	Ω	
P-H-RT	Ω	Ω	Ω	Ω	PH	Ω
P-H-RT-LT	Ω	Ω	Ω	Ω	Ω	CR
P-G	Ω	Ω	LT'	Ω	Ω	Ω
P-H-RT-LT-OK	RL	Ω	Ω	Ω	Ω	Ω

Fig. 4. Primary Table (*PT*) for Room Reservation Component

	P	H	G	RT	LT	OK
ε	(PAR,{Htn,YR}) (LDN,{Sh,Vil})	Ω	Ω	Ω	Ω	Ω
P	Ω	(Htn,{Sgl}) (Sh,{Sgl,Dbl})	(YR, {Sgl,Dor})	Ω	Ω	
P-H	Ω	Ω	Ω	(Sgl,{Std,Dlx}) (Dbl,{Std,Exe})	Ω	Ω
P-G	Ω	Ω	Ω	(Sgl,{Std,Dlx})	Ω	Ω
P-H-RT	Ω	Ω	Ω	Ω	(Std,50$) (Dlx,70$)	Ω
P-H-RT-LT	Ω	Ω	Ω	Ω	Ω	Ω
P-H-RT-LT-OK	(PAR,{Htn,YR})	Ω	Ω	Ω	Ω	Ω

Fig. 5. Auxiliary Table (*AT*) corresponding to Primary Table in figure 4

4.3 Travel Agent Component

The component N of the web application is a Travel Agent. On one side, it accepts inputs from the user and on the other side, it communicates with some back end system. The simplified behavior of the Travel Agent component is as follows. It takes a place name from user and transmits it to the back end. Later, it takes the list of residences from the back end and displays it to the user. The user inputs one of the residences, i.e., Hotel or Guest House, which is then transmitted to the back end. The back end responds with the list of room types for the provided residence. The component selects and resends one of the room types to the back end which then provides the list of luxury types associated with that room type. Once the luxury type is selected, the Travel Agent asks the back end for its price. It shows the corresponding price to the user after increasing it by 10% for its commission. When the user selects OK, the component asks the back end for confirmation, and finally the confirmation message is sent to the user.

The unit testing of Travel Agent component is performed with $I_N^{(1)} = \{UI\_P, RL, UI\_H, UI\_G, RT', LT', PH, PG, UI\_OK, CR\}$. The symbols starting with "UI_" are inputs from user, and symbols starting with "UO_" are outputs to user. A conjecture in the Figure 6 (left) is made when observation tables of the component are found balanced, closed and consistent. The input parameters used in the unit testing and their corresponding output parameters are also given in the figure.

4.4 Integration Testing

In the integration testing, the two components are connected to each other. In this case, all the inputs of component Room Reservation come from component Travel Agent. The component Room Reservation is considered as a back end for Travel Agent which also accepts the inputs from the user. The integration of the learned models of the two components is shown in the Figure 6.

In the procedure, the test input $P(LDN)$ is given to the component Travel Agent, which transmits it to the component Room Reservation. The component produces a list of residences as $RL(\{Htn, YR\})$ and sends back to Travel Agent component, which shows the list to user. The user selects a guest house YR from the list and provides a second input to the integrated system. The component Travel Agent then sends input $G(YR)$ to the component Room Reservation, and it continues working according to the models in the Figure 6. According to the real component of Room Reservation in the Figure 3, the output sequence produced from an input sequence $P(PAR) - G(YR) - RT(Sgl) - LT(Std)$ is $RL(\{Htn, YR\}) - RT'(\{Sgl, Dor\}) - LT'(\{Std, Dlx\}) - PG(50\$)$, but the learned model does not show the output symbol PG. Hence, a divergence is found between the real component and its conjecture. In this case, the above input sequence is treated as a counterexample for Room Reservation component which will be learned again with the help of unit testing. When the observation tables are found balanced, closed and consistent, a new conjecture is made which is equivalent to that in the Figure 3. The conjecture is then integrated with Travel Agent component to complete integration testing.

Apart from the counterexample explained above, the example has other counterexamples with respect to the parameter values. For instance, the component Travel Agent contains some parameter values which can be input from the component Room Reservation, but the learned parameter functions of component Room Reservation are unable to produce those values. As an example, the component Travel Agent expects a guest house named Vil from user, when the list of residences from component Room Reservation is provided. The testing proceeds with the list of room types provided from component Room Reservation, from which $RT(Dbl)$ is selected by the component Travel Agent. The response from the actual component of Room Reservation can be $RT'(\{Std, Stu\})$, which is seen in the output parameter function q of the learned model of Travel Agent component, but function v of the learned model of Room Reservation component is unable to produce. This is because, the component Room Reservation is never tested with residence Vil in its unit testing. Thus, the input sequence $P(LDN) - G(Vil) - RT(Dbl)$ can be treated as a counterexample for this component. In the following unit testing, observation tables will be updated and a new conjecture will be made.

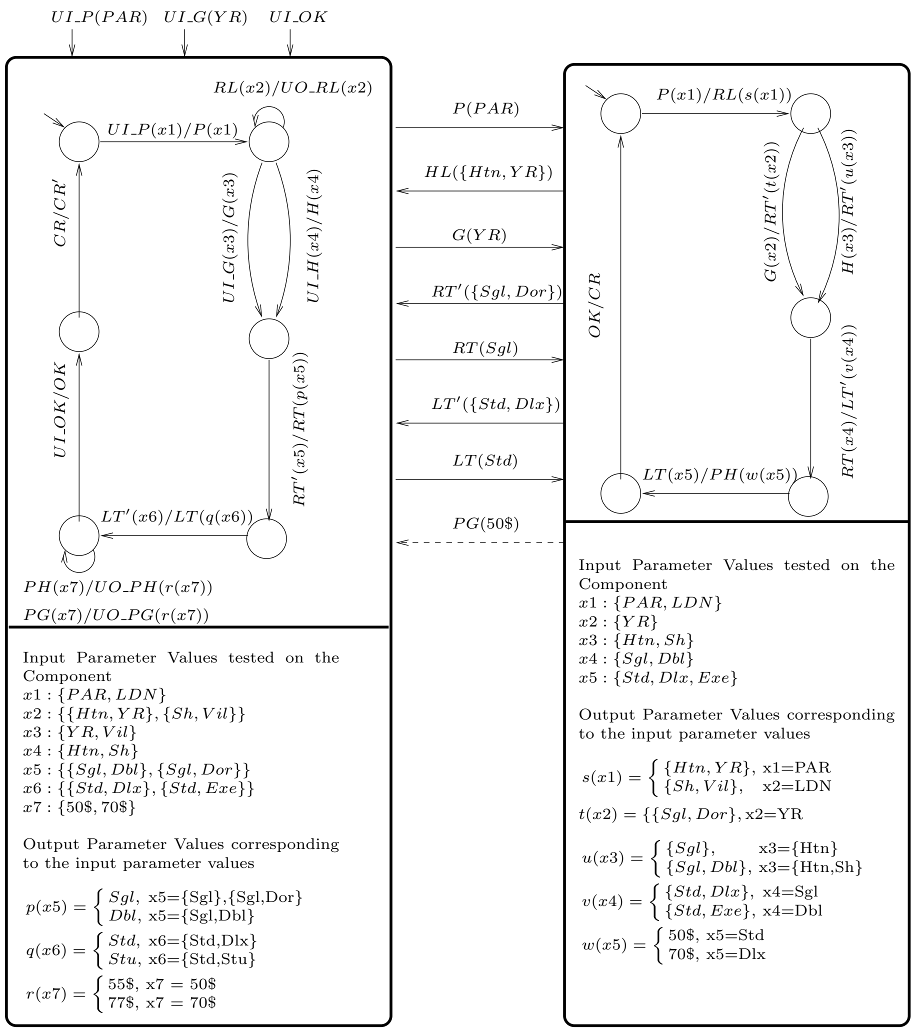

$UI\_P(PAR)$
$UI\_G(YR)$
UI_OK
$RL(x2)/UO\_RL(x2)$
$UI\_P(x1)/P(x1)$
CR/CR'
$UI\_G(x3)/G(x3)$
$UI\_H(x4)/H(x4)$
UI_OK/OK
$RT'(x5)/RT(p(x5))$
$LT'(x6)/LT(q(x6))$
$PH(x7)/UO\_PH(r(x7))$
$PG(x7)/UO\_PG(r(x7))$
Input Parameter Values tested on the Component
$x1 : \{PAR, LDN\}$
$x2 : \{\{Htn, YR\}, \{Sh, Vil\}\}$
$x3 : \{YR, Vil\}$
$x4 : \{Htn, Sh\}$
$x5 : \{\{Sgl, Dbl\}, \{Sgl, Dor\}\}$
$x6 : \{\{Std, Dlx\}, \{Std, Exe\}\}$
$x7 : \{50\$, 70\$\}$
Output Parameter Values corresponding to the input parameter values
$p(x5) = \begin{cases} Sgl, & \text{x5=\{Sgl\},\{Sgl,Dor\}} \\ Dbl, & \text{x5=\{Sgl,Dbl\}} \end{cases}$
$q(x6) = \begin{cases} Std, & \text{x6=\{Std,Dlx\}} \\ Stu, & \text{x6=\{Std,Stu\}} \end{cases}$
$r(x7) = \begin{cases} 55\$, & x7 = 50\$ \\ 77\$, & x7 = 70\$ \end{cases}$
$P(PAR)$
$HL(\{Htn, YR\})$
$G(YR)$
$RT'(\{Sgl, Dor\})$
$RT(Sgl)$
$LT'(\{Std, Dlx\})$
$LT(Std)$
$PG(50\$)$
$P(x1)/RL(s(x1))$
OK/CR
$G(x2)/RT'(t(x2))$
$H(x3)/RT'(u(x3))$
$RT(x4)/LT'(v(x4))$
$LT(x5)/PH(w(x5))$
Input Parameter Values tested on the Component
$x1 : \{PAR, LDN\}$
$x2 : \{YR\}$
$x3 : \{Htn, Sh\}$
$x4 : \{Sgl, Dbl\}$
$x5 : \{Std, Dlx, Exe\}$
Output Parameter Values corresponding to the input parameter values
$s(x1) = \begin{cases} \{Htn, YR\}, & \text{x1=PAR} \\ \{Sh, Vil\}, & \text{x2=LDN} \end{cases}$
$t(x2) = \{\{Sgl, Dor\}, \text{x2=YR}$
$u(x3) = \begin{cases} \{Sgl\}, & \text{x3=\{Htn\}} \\ \{Sgl, Dbl\}, & \text{x3=\{Htn,Sh\}} \end{cases}$
$v(x4) = \begin{cases} \{Std, Dlx\}, & \text{x4=Sgl} \\ \{Std, Exe\}, & \text{x4=Dbl} \end{cases}$
$w(x5) = \begin{cases} 50\$, & \text{x5=Std} \\ 70\$, & \text{x5=Dlx} \end{cases}$

Fig. 6. Integration of PFSM Models of Travel Agent (left) and Room Reservation (right) Components - The dotted line between the components shows missing output from Room Reservation Component

5 Conclusion

In this paper, we propose to use automata learning algorithms as a means to alleviate the absence of models for components, in a model-based testing approach. As in previous related work [3,4], we adapt Angluin's algorithm [1] to a testing context, in an incremental approach. Our contribution extends this approach in two directions. First, we use the models to drive integration testing. Since the models are derived from testing observations, they cannot by themselves constitute oracles or a sound basis for the generation of tests for the components learnt. But they are used to drive the tests of component interactions: partial models learnt for isolated components provide a convenient abstraction that can be used as a basis for covering the sequences of component interactions. Secondly, we extend the learning algorithm to a model where we deal with parameterized inputs and outputs. This is motivated by the fact that actual values exchanged during interactions are a major source of interoperability problems between components.

[2] proposes an extension to Angluin's algorithm where actions can also have parameters, represented by a combination of boolean values. Our algorithm does not include any bound on the domain of parameters, and we introduce the notion of auxiliary table in the algorithm to deal with it without having to explore all combinations. However, in contrast, our model does not include guards on parameters. We are currently working on an extension to include predicates on parameters to trigger different transitions. We are also working towards an implementation of these algorithms to adapt to the practical problems of testing telecommunication services which provided our framework.

Acknowledgement

We would like to thank Alexandre Petrenko (CRIM) for our fruitful discussions on this topic.

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. *Lecture Notes in Computer Science*, 3922:107–121, March 2006.
3. Edith Elkind, Blaise Genest, Doron Peled, and Hongyang Qu. Grey box checking. In *26th IFIP WG 6.1 International Conference on Formal Models for Networked and Distributed Systems (FORTE 2006)*, 2006.
4. Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-specific optimization in automata learning. In *CAV*, pages 315–327, 2003.
5. Keqin Li, Roland Groz, and Muzammil Shahbaz. Inference of parameterized finite state machine - technical report. Technical report, Laboratoire Logiciels Systèmes Réseaux, http://www-lsr.imag.fr/Les.Groupes/VASCO/publi-2006.htm, 2006.
6. Keqin Li, Roland Groz, and Muzammil Shahbaz. Integration testing of components guided by incremental state machine learning. In *Testing: Academic & Industrial Conference - Practice And Research Techniques (TAIC PART)*, 2006.
7. Erkki Mäkinen and Tarja Systä. Mas - an interactive synthesizer to support behavioral modelling in uml. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 15–24, Washington, DC, USA, 2001. IEEE Computer Society.
8. Stephane S. Somé. Beyond scenarios: generating state models from use cases. In *Proceedings of SCESM'02*, 2002.