Integration Testing of Components Guided by Incremental State Machine Learning

Keqin Li
CNRS LSR-IMAG
Keqin.Li@imag.fr

Roland Groz
INPG LSR-IMAG
Roland.Groz@imag.fr

Muzammil Shahbaz
France Télécom
Muhammad.MuzammilShahbaz@orange-ft.com

Abstract

The design of complex systems, e.g., telecom services, is nowadays usually based on the integration of components (COTS), loosely coupled in distributed architectures. When components come from third party sources, their internal structure is usually unknown and the documentation is insufficient. Therefore, the system integrator faces the problem of providing a required system assembling COTS whose behaviour is barely specified and for which no model is usually available.

In this paper, we address the problem of integration testing of COTS. It combines test generation techniques with machine learning algorithms. State-based models of components are built from observed behaviours. The models are alternatively used to generate tests and extended to take into account observed behaviour. This process is iterated until a satisfactory level of confidence in testing is achieved.

1. Introduction

The design of software systems, typically new services offered on the Internet, is more and more based on the integration of components from third party sources (COTS), loosely coupled in a distributed architecture. The system integrator is in charge of providing a new service based on the integration of such components, with minimal development of interfacing software (often nicknamed as "glue"). The integrator faces the problem of providing a required system assembling COTS of which he (she) has a limited knowledge.

Since components come from third party, their internal structure is usually unknown and the documentation is not sufficient to work out all the details of their interactions with other components. Therefore, the integrator would typically first test each component to learn its behaviour on typical requests it would have to serve in the assembly, and then test the integrated system based on variations of system use cases.

Due to the lack of formal models of components, in order to design test cases for the components and the integrated system, the integrator has to rely on his intuitions. Currently, there are not many tools that can help him in test generation.

At the same time, use cases are often provided by domain experts (including system users). The use cases might be informal and obtained in an ad hoc way, and thus are hardly sufficient to check that the components will interact correctly in any combination of requests.

On the other hand, test generation tools for formal specifications based on state machines [5] could offer some help in system integration; thus, formal models of components are needed.

Our work addresses this key issue of providing a support to component integration in this context. In order to systematically develop tests with a satisfactory level of confidence, we use test generation techniques based on formal models. Since formal models cannot be expected to be delivered with COTS, we shall infer models from the observed behaviours during tests. This is the key point in our approach. Our basic assumptions are as follows.

- Components will be seen as black boxes. Their interfaces are known. This means that we know at least partial input and output types.
- 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.
- Inputs from the environment will only be provided on stable states of the system, viz. when the system is waiting for external stimuli and will not make any internal move. This corresponds to the assumption of slow environment in system verification and testing. We are assuming that the system has a reactive semantics.

- Some test scenarios can be provided by domain experts as a guideline for the system integration. Test scenarios are expressed in terms of externally observable actions of the integrated system.

In the absence of formal models for components, model inference from scenarios is a key point. We propose an extension of Angluin's algorithm [1] that fits into that framework. The original algorithm has been proposed for Deterministic Finite-State Acceptors (DFAs) as language acceptors. Most uses of this algorithm for I/O machines have reused the algorithm with a simple mapping from inputs and outputs to single letters into a DFA's alphabet A: either by taking inputs and outputs as letters: $A=I\cup O$ as in [2][4], or by considering couples of inputs and outputs as letters: $A=I\times O$ [7]. Our extension uses only inputs as letters of the alphabet A of Angluin's algorithm, and outputs are used in the cells of the table instead of booleans indicating the membership. Finally, the trickiest ingredient of the initial algorithm, namely the equivalence query, finds a practical implementation in our framework. Contrary to requirements engineering approaches such as [7] or [8], the equivalence query is not provided by an expert, but by testing the system which acts as an oracle.

The overview of the integration testing methodology is as follows.

1. In the first step, an input alphabet is defined for each component C starting with those from use cases defined for the system.
2. The components are integrated, which means that some of the outputs of one component will appear as inputs on the connected interface of another component. The behaviour of the integration system is tested according to test scenarios. If the test scenarios are not respected, the problematic component is identified and replaced. At the same time, the observed behaviour of each component C is recorded in its Observation Table T_C. The procedure is referred as Scenario Testing in the following.
3. Starting from its observation table, each component is tested separately using the learning algorithm until a closed and consistent table is found. The output alphabet is determined. This provides the first model $C^{(1)}$ for C. The procedure is referred as Unit Testing in the following.
4. The components are integrated again, and the whole system is tested based on the models of the components according to a certain test generation strategy. A chosen strategy is built on some model coverage criterion. The actual outputs, both, internal and external, observed are recorded and compared to those provided by the models. Tests are performed until a discrepancy between predictions from the model and the observed outputs is found or the criterion is achieved. The procedure is referred as Integration Testing in the following.
5. In case of discrepancy, the model has to be corrected, so we extend the models by relearning and iterate Step 4 with the new $C^{(i+1)}$ components.
6. When the coverage criterion is achieved, domain experts check the component models and test results. In case unexpected behaviours have been identified, components may be replaced, and Unit Testing and Integration Testing are iterated.
7. The process terminates after domain experts approve the results.

This incremental approach is similar to the one presented in [3]. In this paper, we concentrate on the use of such an approach for component integration, and develop the necessary extensions to the learning algorithm.

The rest of the paper is organized as follows. We present our basic model and data structure in the next section. After introducing integration testing architecture in Section 3, Scenario Testing, Unit Testing and Integration Testing are presented in Section 4, 5, and 6, respectively. An example is presented in Section 7. Finally, Section 8 concludes the paper and describes possible future works.

2. Preliminary

In this section, we give the definition of FSM, which is the basic model we use, and an observation table, which is the basic data structure, where the information observed in testing is recorded.

2.1. Finite State Machine

A *Finite State Machine* (FSM) M is a six-tuple $M=(Q, I, O, \delta, \lambda, q_0)$, where Q, I, O are finite and nonempty sets of states, input symbols, and output symbols, respectively. $\delta:Q\times I\rightarrow Q$ is the state transition function. $\lambda:Q\times I\rightarrow O$ is the output function. $q_0\in Q$ is the initial state. When the FSM is in a current state q in Q and receives an input a from I, it moves to the next state specified by $\delta(q,a)$, and produces an output given by $\lambda(q,a)$.

We extend the transition function δ and the output function λ from input symbols to strings as usual: for a state q_1, an input sequence $x=a_1,\ldots,a_k$ takes the FSM successively to states $q_{i+1}=\delta(q_i,a_i)$, $i=1,\ldots,k$, with the final state $\delta(q_1,x)=q_{k+1}$, and produces an output sequence $\lambda(q_1,x)=b_1,\ldots,b_k$, where $b_i=\lambda(q_i,a_i)$, $i=1,\ldots,k$.

We consider only *input enabled* FSM, that is when $dom(\delta)=dom(\lambda)=Q\times I$. A machine can be made input enabled by adding loopback transitions with a special output symbol "Ω". "Ω" is an abstraction for an explicit invalid notification from the system.

2.2. Observation Table

In the work, we model a component C as an unknown FSM $M=(Q, I, O, \delta, \lambda, q_0)$ with known input symbols I. Since we can submit any input sequence to the component and observe the corresponding output sequence, for any $\alpha\in I^*$, $\lambda(q_0, \alpha)$ is known. We also assume that each component can be reset to its initial state before each test.

In the testing procedure, the observed behaviour of the component C is recorded into its *Observation Table*, using a nonempty finite prefix-closed set S of input strings (representing potential states of the FSM), a nonempty finite suffix-closed set E of input sequences (separating potential states of the FSM) $(\varepsilon\notin E)$, and a finite function T mapping $((S\cup S\cdot I)\times E)$[1] to O^*. The observation table is denoted by (S, E, T).

In the observation table, for each $\sigma\in(S\cup S\cdot I)$, $\mu\in E$, $T(\sigma,\mu)=\beta$, $\beta\in O^*$, such that $|\beta|=|\mu|$, and $\lambda(q_0, \sigma\cdot\mu)=\lambda(q_0, \sigma)\cdot\beta$.

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

An observation table can be visualized as a two–dimensional array with rows labelled by elements of $(S\cup S\cdot I)$ and columns labelled by elements of E, with the entry for row s and column e equal to $T(s, e)$. If s is an element of $(S\cup S\cdot I)$, $row(s)$ denotes the finite function f from E to O^* defined by $f(e)=T(s, e)$. An example of the observation table is given in Table 1. In this observation table, $I=\{a, b\}$, $O=\{x, y\}$, $S=\{\varepsilon, a, b\}$, $E=\{a, b\}$.

An observation table is called *closed* provided that for each t in $S\cdot I$ there exists an s in S such that $row(s)=row(t)$. An observation table is called *consistent* provided that whenever s_1 and s_2 are elements of S such that $row(s_1)=row(s_2)$, for all a in I, $row(s_1\cdot a)=row(s_2\cdot a)$.

If (S, E, T) is a closed, consistent observation table, we define an FSM $M(S,E,T)=(Q', I, O, \delta', \lambda', q'_0)$ in which:

$$Q'=\{row(s):s\in S\},$$
$$q'_0=row(\varepsilon),$$
$$\delta'(row(s),a)=row(s\cdot a),\ a\in I$$
$$\lambda'(row(s),a)=T(s, a),\ a\in I.$$

[1] Empty sequence is denoted by ε. Concatenation of strings and their sets is denoted by "·".

For example, the observation table in Table 1 is closed and consistent. We depict the corresponding $M(S, E, T)$ in Figure 1. In it, $Q'=\{row(\varepsilon), row(a), row(b)\}$.

Table 1. An example observation table

	a	b
ε	x	x
a	x	y
b	y	x
aa	x	y
ab	y	x
ba	x	y
bb	y	x

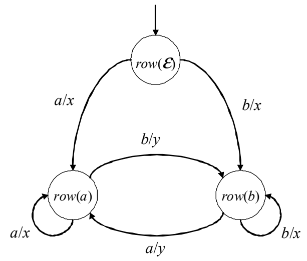

Figure 1. Conjecture machine

For the FSM $M(S, E, T)$, we have the following theorems:

Theorem 1. Assume that (S, E, T) is a closed and consistent observation table, then the FSM $M(S, E, T)$ is consistent with the finite function T. That is, for every s in $(S\cup S\cdot I)$ and e in E, $\lambda'(q'_0, s\cdot e)=\lambda'(q'_0, s)\cdot T(s, e)$.

Theorem 2. Assume that (S, E, T) is a closed and consistent observation table, any other FSM consistent with T, but inequivalent to $M(S, E, T)$ must have more states.

The proof of these theorems follows Angluin's [1], and is omitted here due to page limit.

3. Integration Testing Architecture

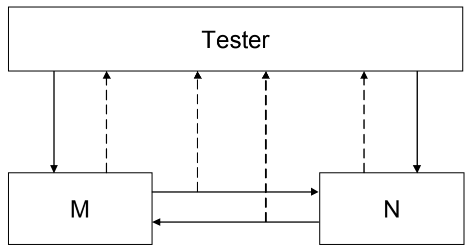

Figure 2. Integration testing architecture

The integration testing architecture for a system of two components M and N is illustrated in Figure 2: for each component, there are internal interfaces to connect it to another component, and external interfaces to connect it to the tester or the environment. Accordingly, the symbols transmitted through the two kinds of interfaces are called *internal symbols* and *external symbols*, respectively. For example, in the component M, I_M can be divided into the set of external input symbols EI_M and the set of internal input symbols II_M, i.e., $I_M=EI_M \cup II_M$ and $EI_M \cap II_M=\emptyset$.

Through the external interfaces, the tester submits external input symbols and observes external output symbols. At the same time, we assume that through the internal interfaces, the tester can observe the interactions between the components. Thus, the behaviour of any given component is observable.

In the integration testing procedure, the external input is given to the integrated system only when no internal transition is possible. Moreover, we assume that the system has a single message in transit, i.e., for each component and each input, only one output is produced. Thus, a one place buffer between communicating components suffices.

4. Scenario Testing

In integration testing, normally we have some test scenarios which the integrated system is supposed to implement. A test scenario is an input/output sequence containing external inputs and outputs.

For each test scenario, a test case is constructed. When executing a test case, we submit an external input symbol to the integrated system, observe the external output symbol. At the same time, by observing the internal interfaces, we also obtain input/output sequences of every component in the system.

After executing the test case, we check whether the test scenario has been respected. If not, we identify the problematic component (that provides the first diverging output), replace it and start all over again.

For the observed behaviour of a single component, e.g., M, we record the information in its observation table T_M. Suppose the input sequence is $\alpha=i_1,i_2,\ldots,i_k$, $i_j \in I_M$ $(1 \le j \le k)$, and the corresponding output sequence is $\beta=o_1,o_2,\ldots,o_k$, $o_j \in O_M$ $(1 \le j \le k)$. We add all the prefixes of α to the set S, and record o_j $(1 \le j \le k)$ in the corresponding cells of the observation table, i.e., $T(i_1,i_2,\ldots,i_{j-1}, i_j)= o_j$.

Table 2. Observation table

	a	b
ε	x	
a		x
ab		y
abb		

As an example, assume that $I_M=\{a,b\}$, $O_M=\{x, y\}$, the input sequence is abb, and the corresponding output sequence is xxy. The corresponding observation table T_M is shown in Table 2.

The observation tables of all components are filled based on observations made during testing all scenarios.

5. Unit Testing

After the scenario testing, for each component C, some cells have been filled in its observation table T_C. Then we begin the unit testing in which each component is tested and learned individually.

The first step in unit testing is to complete the initial observation table. For each s in $S \cup S \cdot I$, e in E, if $T(s, e)$ is not yet known, we perform a test using $s \cdot e$ as the input sequence, and obtain the corresponding output sequence $x \cdot y \in O^*$, in which $|x|=|s|$ and $|y|=|e|$. Thus, $T(s, e)=y$.

We continue testing using the extended learning algorithm until the observation table is closed and consistent. Then a conjecture model $C^{(1)}$ is made.

5.1. Extended Learning Algorithm

Similar to Angluin's learning algorithm, in unit testing, we check whether the observation table (S, E, T) is closed and consistent.

If (S, E, T) is not closed, we find s_1 in S and a in I such that for all s in S, $row(s_1 \cdot a) \ne row(s)$. We add the string $s_1 \cdot a$ to S so that S becomes S' and extend T to $((S' \cup S' \cdot I) \times E)$ by testing for missing elements.

If (S, E, T) is not consistent, we finds s_1 and s_2 in S, e in E, and a in I such that $row(s_1)=row(s_2)$, but $T(s_1 \cdot a \cdot e) \ne T(s_2 \cdot a \cdot e)$. We add the string $a \cdot e$ to E so that E becomes E' and extend T to $((S \cup S \cdot I) \times E')$ by testing for missing elements.

When the observation table is closed and consistent, the conjecture $M(S, E, T)$ is made, and instead of equivalence query in Angluin's algorithm, we finish the learning algorithm.

This is a major difference from Angluin's algorithm and the implementation of it in state model synthesis from scenarios [7]. We do not rely on an expert to answer this query. Instead, we use the black box system as an oracle: the equivalence query will be implemented by a testing strategy, which will provide counterexamples. In particular, we generate (conformance) tests using conjectured models and if the observed behaviour deviates from that of available models then we need to refine them. Note that this is also the converse of what is usually considered an oracle in model-based testing: here the oracle is the system, not the model.

For the termination of the algorithm, we have the following theorem:

Theorem 3. Let (S, E, T) be an observation table. Let n denote the number of different values of $row(s)$ for s in S. Any FSM consistent with T must have at least n states.

Suppose component C has n states, according to Theorem 3, the number of different values of $row(s)$ for s in S in observation table T_C cannot be more than n. Based on the operations used in the algorithm, we can prove that the number of different values of $row(s)$ increases monotonically. So, similar to the original algorithm [1], the algorithm always eventually finds a closed and consistent observation table and makes a conjecture.

5.2. Dealing with I/O Counterexample

After a conjecture model $M(S, E, T)$ is made for a component M, in the integration testing, we may find that for a certain input sequence, the output of $M(S, E, T)$ may be different from the output of component M. The input sequence is considered as a *counterexample*.

When a counterexample is found, it can be used to extend the observation table (S, E, T) and to make a new conjecture. We proceed as follows.

For each counterexample t, t and all its prefixes are added to S, so that S becomes S'. Then the function T is extended to $((S' \cup S' \cdot I) \times E)$ by testing for the missing elements.

After that, following the procedure of the extended learning algorithm, the observation table (S, E, T) is made closed and consistent. Finally, a new conjecture is made based on it.

5.3. Dealing with New Input Symbols

Another motivation of extending observation table and making a new conjecture for a component is that some new input symbols could have been triggered during integration as a result of output from another component. Normally they could be discovered either when unit testing another component, or when integration provides a counterexample. But for the first case, we only need to take them into account when they are actually exercised in integration testing.

When new input symbols have been identified, they are added to I to obtain I'. At the same time, they are added to E to obtain E'. Then the function T is extended to $((S \cup S \cdot I') \times E')$ by testing for the missing elements.

After that, following the procedure of the extended learning algorithm, the observation table (S, E, T) is made closed and consistent. Finally a new conjecture is made based on it.

6. Integration Testing

At the end of unit testing, a conjecture FSM is obtained for each component. Then the integration testing procedure begins.

In this stage, the components are integrated, and their joint behaviour is 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. After the unit testing (learning) of them, the initial models $M=(Q_M, I_M, O_M, \delta_M, \lambda_M, q_{M0})$ and $N=(Q_N, I_N, O_N, \delta_N, \lambda_N, q_{N0})$ are constructed.

6.1. Integration Testing Procedure

In integration testing, test cases are constructed according to some test generation strategy. A chosen strategy is usually built on some coverage criterion. The existing interoperability testing methods, e.g., [10], can be adapted here. In this work, we present an approach based on the composition machine of FSMs.

Whenever a test case has been generated, we execute it. We check whether the observed behaviour conforms to the models of components, and go back to the unit testing procedure whenever a counterexample has been found.

This stage and thus the integration testing procedure terminate when the chosen coverage criterion is satisfied and domain experts have approved the testing results.

6.2. Test Generation based on Composition of FSMs

In [12], the composition of FSMs is defined as follows. A system of two components includes interconnected FSMs M and N, and an environment E that submits a next external input to the system only after the system has produced an external output in response to the previous input. E is described by a Label Transition System (LTS) L_E with two states. The behaviour of the closed system can be described by the LTS composition $L_M \| L_N \| L_E$, where L_M and L_N are LTSs corresponding to FSMs M and N.

Given the LTS $L_M \| L_N \| L_E$ without livelocks in the sense of [12], we determine the external projection of the LTS $L_M \| L_N \| L_E$ onto the external alphabets and transform the projection into an FSM, denoted $M \Diamond N$, by pairing each input with a subsequent output and replacing the pair of corresponding transitions by a single transition.

In [13] an algorithm is provided to compute $M\Diamond N$. Each transition in $M\Diamond N$ is derived from either a transition of only one component or a combination of transitions of the two components.

In [13], the purpose is to generate test sequences to test an embedded component (either M or N). Therefore, global transitions that include local transitions of the embedded component are considered.

In our work, in order to test the interactions of two components, we ignore global transitions that are in fact single local transitions of a particular component, as they do not trigger any component interaction. We consider global transitions that are derived from a combination of transitions of the two components. Then, we assign colour to these global transitions, and use the algorithm proposed in [11] to find a (minimal) number of paths from the initial state of $M\Diamond N$, where each path covers a maximum number of not covered yet colours. Each path is treated as one test, and the set of tests cover all the coloured transitions.

In this way, a set of integration tests is generated and executed to uncover the behaviour of the system in addition to that prescribed by the given scenarios.

6.3. Dealing with Discrepancy

When executing integration tests, we observe the behaviour of each component and compare with its model to detect discrepancy.

When discrepancy between a component and its model, say M, has been found, the integration testing stops.

There are two possibilities of discrepancy:

- An input sequence $x \in I_M^*$ produces an unexpected output sequence.
- The other component N produces an output symbol $a \notin I_M$ as an unexpected input to M.

In the first case, x is a counterexample for M, we go back to unit testing, and follow the process described in Section 5.2.

In the second case, N produces a new input symbol to M, we go back to unit testing, and follow the process described in Section 5.3.

After the new round of unit testing, the observation table and thus the model M are updated.

6.4. Consulting Domain Experts

At the end of integration testing, the models of components and the test results for all the executed tests are presented to domain experts. Based on this information, domain experts may:

- Identify unexpected behaviours, either directly from the models of components, or from test scenarios that do not meet his/her expectations.
- Propose additional test scenarios, if it is realized that some expected behaviours have not been tested.

With the feedbacks from domain experts, the components may need to be retested, the models updated, and/or components may need to be replaced.

The whole process terminates when domain experts are satisfied with the integration testing results. Note that at this point we also have an objective assessment of the quality of the tests based on the coverage reached at the end of the process.

6.5. Result of Integration Testing

At the end of integration testing, for each component, we have a state machine model, which is consistent with all the tests that have been passed. As stated in Theorem 2, if the component and the model have the same numbers of states, they are equivalent. At the same time, the joint behaviour of these components has been systematically tested. Using the approach based on composition of FSMs, a transition coverage by tests is achieved.

Last, but not least, execution of integration tests could reveal some faults related to individual components or the way they are integrated. Starting with a limited input from the designer (viz. identification of interfaces plus a set of scenarios), our approach systematically tests the combinations of external inputs to the system to expose unintended interactions.

7. Example

Assume the integrator is developing a travel agency web application, in which two components are identified: Hotel Reservation and User Interface.

7.1. Component Hotel Reservation

The simplified working procedure of the component Hotel Reservation is as follows: first, the user of the component inputs his/her name, and the component provides the user with a hotel list. After the user selects one from two hotels, the component provides him/her with the types of bedrooms: double or single. After the user selects a type, a form for indicating a time period is provided. When this is done, the corresponding price is provided. At the end, the user selects *OK* to confirm, or selects *No* to reselect.

The component can be described by an FSM $M=(Q_M, I_M, O_M, \delta_M, \lambda_M, q_{M0})$, shown in Figure 4. For simplicity, some transitions have not been depicted. For each state, if there is no transition for some input in Figure 4, the machine outputs Ω and remains in the same state.

In this example, we start with empty test scenarios, so we skip the first step (Scenario Testing).

7.2. Unit Testing of Component Hotel Reservation

In the unit testing procedure, the component Hotel reservation is considered as a black box, no initial model is available, and I_M is known as {*N*, *H1*, *H2*, *Dbl*, *Sgl*, *T*, *OK*, *No*}. The learning procedure is described as follows.

Initially $S=\{\varepsilon\}$, $E=I_M$. The tester applied several input sequences to construct the initial observation table which is not closed, since *row*(*N*) is not equivalent to *row*(*ε*).

The tester adds *N* to *S* and executes more input sequences to construct the observation table T_2 which is shown in Table 3.

This observation table is still not closed, since *row*(*N-H1*) is not equivalent to any *row*(*s*) for $s\in S$. The tester adds *N-H1* to *S* and continues the unit testing procedure until the observation table is closed and consistent. Then, an FSM *M*(*S*, *E*, *T*) is conjectured, which is shown in Figure 5.

7.3. Component User Interface

Now we consider the component of the web application called User Interface, which is the front end of the system. On one side, it accepts user input, such as clicking button or selecting item from drop list, converts the input to corresponding message, and sends the message to back end. On the other side, it accepts messages from back end, and presents their content in a proper format, such as a static text or list, to the user.

The component user Interface is shown in Figure 3, in which the symbols starting with "*UI_*" are inputs from the user, and symbols starting with "*UO_*" are outputs to the user. In this example, we assume that at the end of unit testing of this component, the exact model has been constructed.

7.4. Integration Testing

In the integration testing, the two components are connected. In this case, all the inputs of the component Hotel Reservation come from the component User Interface. So test sequences are generated and executed on User Interface.

In the procedure, the test sequence (*UI_Name*, *UI_H2*, *UI_Dbl*, *UI_T*) is an input to User Interface. The actual input sequence of the component Hotel Reservation is (*N*, *H2*, *Dbl*, *T*), and the corresponding output sequence is (*HL*, *TL*, *F*, *PD2*). But according to the model of Hotel Reservation, the corresponding output sequence should be (*HL*, *TL*, *F*, *PD1*). An unexpected output sequence occurs. Then, (*N*, *H2*, *Dbl*, *T*) is considered as a counterexample, and the model of component Hotel Reservation is refined.

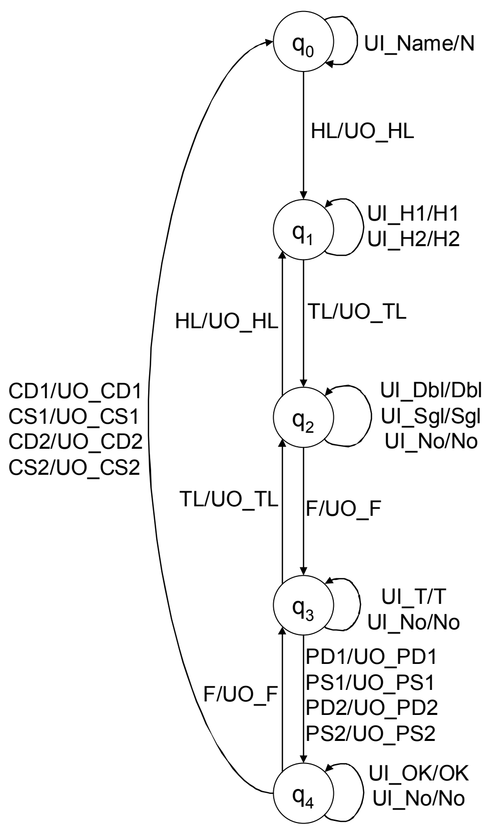

Figure 3: Component User Interface

When the observation table is closed and consistent, the new conjecture of component Hotel Reservation is equivalent to that in Figure 4. In the following integration testing, no discrepancy is found. The integration testing terminates.

It should be emphasized that the observed interactions have served as counterexamples for refining the models of individual components.

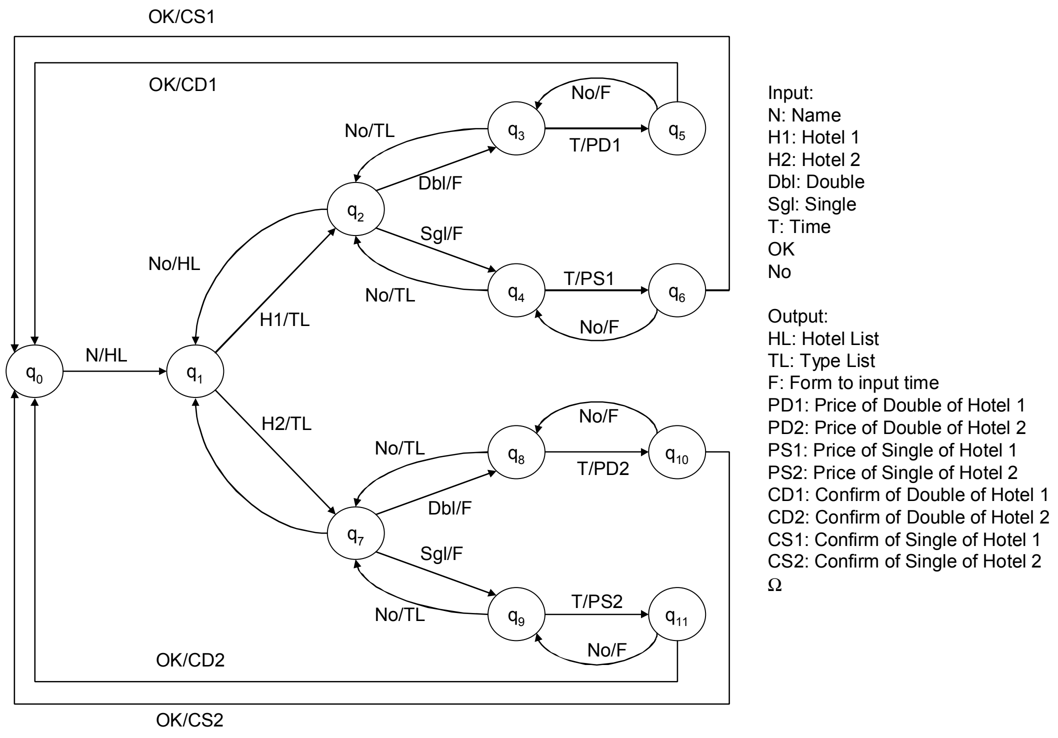

Figure 4: Component Hotel Reservation

Table 3. Observation table T_2 after adding *N* to *S*

T_2	*N*	*H1*	*H2*	*Dbl*	*Sgl*	*T*	*OK*	*No*
ε	HL	Ω	Ω	Ω	Ω	Ω	Ω	Ω
N	Ω	TL	TL	Ω	Ω	Ω	Ω	Ω
H1	HL	Ω	Ω	Ω	Ω	Ω	Ω	Ω
H2	HL	Ω	Ω	Ω	Ω	Ω	Ω	Ω
Dbl	HL	Ω	Ω	Ω	Ω	Ω	Ω	Ω
Sgl	HL	Ω	Ω	Ω	Ω	Ω	Ω	Ω
T	HL	Ω	Ω	Ω	Ω	Ω	Ω	Ω
OK	HL	Ω	Ω	Ω	Ω	Ω	Ω	Ω
No	HL	Ω	Ω	Ω	Ω	Ω	Ω	Ω
N-N	Ω	TL	TL	Ω	Ω	Ω	Ω	Ω
N-H1	Ω	Ω	Ω	F	F	Ω	Ω	HL
N-H2	Ω	Ω	Ω	F	F	Ω	Ω	HL
N-Dbl	Ω	TL	TL	Ω	Ω	Ω	Ω	Ω
N-Sgl	Ω	TL	TL	Ω	Ω	Ω	Ω	Ω
N-T	Ω	TL	TL	Ω	Ω	Ω	Ω	Ω
N-OK	Ω	TL	TL	Ω	Ω	Ω	Ω	Ω
N-No	Ω	TL	TL	Ω	Ω	Ω	Ω	Ω

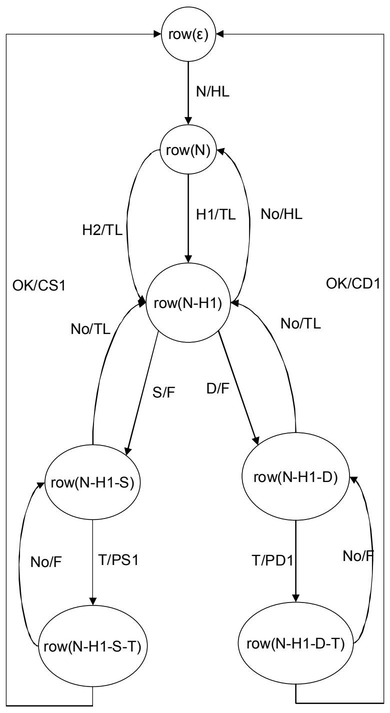

Figure 5. Conjecture of component Hotel Reservation

8. Conclusion

We presented a global approach that combines machine learning and model-based testing for the integration of components whose structure and code is unknown.

This approach reuses Angluin's algorithm [1], while the iterative refinement of models through testing is similar to previous work done on testing telecommunication systems [3], although with different algorithms. Central to our approach is integration of components that is a concern faced by most software developers nowadays. We stop the learning procedure when a conjecture model is made, and move forward to integration testing. In this procedure, interactions between components are tested, and from the point of view of learning algorithm, counterexamples can be found.

In our approach, the test designer just has to concentrate on the abstraction of actual input and output events to consider in the models of components. It is a common observation that software designers can readily identify interfaces and data values, and provide some formal conceptualization or abstraction of them, whereas the real difficulty lies in producing a formal model for the behaviour. Our approach leaves the easiest part (event abstraction from actual PDUs or component interactions) to the test designer, and the model is built automatically based on this provided abstraction.

Actually, in our approach, formal models are just a "by-product" which could be thrown away, or kept for documentation purposes. Contrary to requirements engineering or other approaches in retrieving models from scenarios, such as [7][8][9], our goal is not to build a complete model of the system. The model is just used as a basis for the testing process to:

- Generate integration tests of components.
- Assess the coverage, hence the quality of the integration testing process.

However, integrating our algorithms with other scenario and state-based development environments would provide a convenient link with global engineering approaches.

So far, with a minimal input from the system integrator, namely a preliminary definition of input alphabets in unit testing, our approach provides a sound method for generating integration tests with a measurable level of confidence in testing results.

As other authors who rely on Angluin's algorithm, we have concentrated on FSM models. As the example presented in Section 7 suggests, enumerating input parameter values (such as name of hotels in the example) can lead to replications in FSM structures. So our next step will be to move towards a representation of parameterized inputs and predicates on parameters, i.e., we will work on Parameterized FSM [14]. As one style to extend FSM, with the help of abstraction of parameters, we can represent FSM in a compact way, and with parameter having an infinite domain, we can enrich the expressiveness of FSM. At the same time, Parameterized FSM will pose new challenges on both learning and testing. For example, in integration testing, along with the same input symbol, parameter values that have not been used in unit testing, will be produced by other components in integration. How to select parameters in test generation is another issue.

There is another way to extend FSM, i.e., introducing variables into FSM. The combination of these two ways, and connecting to some sort of Extended Finite State Machines, which could be Statecharts [7] [8] [9] or some other adequate representation, is another work item.

In all this, we are trying to adapt the level of abstraction of the inferred model so as to be able to generate completely instantiated tests for a real system, while keeping a limited size for the models. Of course, the task of inferring a fully accurate model would blow up any implementation of the approach. This is why we are inferring only abstractions, to fight complexity. Those abstractions correspond to an approximation of the actual system, yet this approximation is enough to test the system up to the required level (defined by the coverage and stopping criteria).

One direction for future work will be to come up with a more precise identification of the type of approximation needed.

We shall also be working on experiments and case studies on real size components and systems, e.g., web services, to assess the applicability of this proposed approach, and the types of abstractions needed or better suited to given types of applications.

Another direction for future work is to revisit model-based test generation strategies and the corresponding coverage criteria that would be adapted to our integration approach.

9. Acknowledgement

We would like to thank Alexandre Petrenko (CRIM) for our fruitful discussions on this topic. We would also like to thank the anonymous reviewers for their useful and detailed comments.

10. References

[1] D. Angluin, "Learning Regular Sets from Queries and Counterexamples," *Information and Computation*, 75 (1987), pp. 87-106.

[2] A. Groce, D. Peled, and M. Yannakakis, "Adaptive Model Checking," *Proceedings of TACAS 2002*, 2002, pp. 357-370.

[3] A. Hagerer, H. Hungar, O. Niese, and B. Steffen, "Model Generation by Moderated Regular Extrapolation," *Proceedings of FASE 2002*, LNCS 2306, 2002, pp. 80-95.

[4] H. Hungar, O. Niese, and B. Steffen, "Domain-Specific Optimization in Automata Learning," *Proceedings of CAV 2003*, LNCS 2725, 2003, pp. 315-327.

[5] D. Lee, M. Yannakakis, "Principles and Methods of Testing Finite State Machines – a Survey," *Proceedings of the IEEE*, Vol. 84, No. 8, 1996, pp. 1090-1126.

[6] S. Leue, L. Mehrmann, M. Rezai, "Synthesizing ROOM Models from Message Sequence Chart Specifications," Technical Report 98-06, University of Waterloo, Canada, April 1998.

[7] E. Mäkinen, T. Systä, "MAS – an Interactive Synthesizer to Support Behavioural Modelling in UML," *Proceedings of ICSE 2001*, 2001, pp. 15-24.

[8] S. Somé, "Beyond Scenarios: Generating State Models from Use Cases," *Proceedings of SCESM 2002*, 2002.

[9] J. Whittle, J. Schumann, "Generating Statechart Designs from Scenarios," *Proceedings of ICSE 2000*, 2000, pp. 314-323.

[10] O. Koné, R. Castanet, "Test Generation for Interworking Systems," *Computer Communications*, 23, 2000, pp. 642-652.

[11] M. Kim, C. Besse, A. Cavalli, and F. Zaïdi, "Two Methods for Interoperability Tests Generation: An Application to the TCP/IP Protocol," In *Proceedings of TestCom 2002*, Berlin, 2002.

[12] A. Petrenko, N. Yevtushenko, "Solving Asynchronous Equations," In *Proceedings of FORTE'98, the Eleventh IFIP International Conference on Formal Description Techniques for Distributed systems and Communications Protocols*, France, 1998

[13] L. Lima Jr., A. Cavalli, "A Pragmatic Approach to Generating Test Sequences for Embedded Systems," In *Proceedings of IWTCS'97, The 10th International IFIP TC6/WG6.1 Workshop on Testing of Communication Systems*, Cheju Island, Korea, 8-10 September, 1997

[14] M. Shahbaz, "Incremental Inference of Black-Box Components to Support Integration Testing," In *Testing: Academic & Industrial Conference – Practice And Research Techniques, (TAIC PART 2006)*, PhD Programme, Windsor, UK, 2006