DIAGNOSING MULTIPLE FAULTS IN COMMUNICATING FINITE STATE MACHINES

Khaled El-Fakih+, Nina Yevtushenko++ and Gregor v. Bochmann+
+School of Information Technology and Engineering, University of Ottawa, ON, KIN 6N5, Canada. {kelfakih, bochmann} @ site. uottawa.ca
++Tomsk State University, 36 Lenin Str., Tomsk, 634050, Russia, yevtushenko.rff@elefot.tsu.ru

Abstract In this paper, we propose a method for diagnostic test derivation when the system specification and implementation are given in the form of two communicating finite state machines and at most a single component machine can be faulty. The method enables to decide if it is possible to identify the faulty machine in the system, once faults have been detected in a system implementation. If this is possible, it also provides tests for locating the faulty component machine. Two examples are used to demonstrate the different steps of the method. The method can also be used for locating faults within a machine when the system specification and implementation are given in the form of a single FSM.

Keywords: Diagnostics, conformance testing, communicating finite state machines

1. INTRODUCTION

The purpose of conformance testing is to check whether an implementation conforms to its specification. Usually a conforming implementation is required to have the same I/O behavior. An interesting complementary yet more complex problem is to locate the differences between a specification and its implementation when the implementation is found to be nonconforming [7]. A solution to this problem has various applications. For example, it makes easy the job of correcting the implementation so that it conforms to its specification [7].

In the software domain where a system may be represented as an FSM, some work has already been done for the *diagnostic* and *fault localization*

problems [2][4][7]. However, the work done for distributed systems represented as two Communicating FSMs [3] makes some important simplifying assumptions. In [2][7] and [3] the difference between the system specification and its implementation is located under the assumption of a single fault in the implementation. In [4] the differences can be located for multiple faults under the assumption that each of the faults is reachable through non-faulty transitions.

In this paper, we consider a system consisting of two communicating FSMs, called components. One component, called *context machine,* communicates with the environment and the other component, called *embedded machine*. The interaction between these two components is assumed to be hidden, that is, unobservable. We assume that multiple output or transfer faults may occur in at most one component [5]. None of these faults will increase the number of states in the implementation of the component. It is not always possible to locate the faulty component of the given system when faults have been detected in a system implementation (SUT). This happens when certain faults in an implementation of the context and other faults in the implementation of the embedded component may cause the same observable behavior of the SUT. We present a novel approach for diagnostic test derivation. The approach enables us to decide whether it is possible to identify the faulty component in the given system, and if this is possible then tests for locating the fault are derived. We use a non-deterministic FSM for the compact representation of possible faulty transitions. The same technique can also be used for locating multiple faults when the system specification and implementation are given in the form of a single FSM.

This paper is organized as follows. Section 2 comprises necessary definitions for communicating FSMs. In Section 3, the diagnostic problem is discussed, while the method to solve the problem is presented in Section 5. In Section 4, the different steps of the method are presented illustrated by a working example. Future work is described in Section 6.

2. FINITE STATE MACHINES

A non-deterministic finite state machine (FSM) is an initialized non-deterministic Mealy machine which can be formally defined as follows [9]. A *non-deterministic finite state machine A* is a 6-tuple $\langle S, I, O, h, D_A, s_0\rangle$, where S is a finite nonempty set of n states with s_0 as the initial state; I and O are input and output alphabets; D_A is a specification domain which is a subset of $S\times I$; and h: $D_A \rightarrow 2^{S\times O}\backslash\varnothing$ is a behavior function where $2^{S\times O}$ is the set of all subsets of the set $S\times O$. The behavior function defines the possible transitions of the machine. Given present state *si* and input symbol *i,* each

pair $(s_j,o)\in h(s_i,i)$ represents a possible transition to the next state s_j with the output o. This is also written as a transition of the form $s_i \xrightarrow{i/o} s_j$. If $D_A = S\times I$ then A is said to be a *complete* FSM; otherwise, it is called a *partial* FSM. In the complete FSM we omit the specification domain D_A, i.e. complete FSM is 5-tuple $A = \langle S, I, O, h, s_0\rangle$. If for each pair $si\in D_A$ it holds that $|h(s,i)| = 1$ then FSM A is said to be *deterministic.* In the deterministic FSM A instead of behavior function h we use two functions, transition function $\delta: S\times I \to S$ and output function $\lambda: S\times I \to O$. FSM $B = (S',I,O,g,s_0)$, $S'\subseteq S$, is a *sub-machine* of complete FSM A if for each pair $si\in S'\times I$, it holds that $g(s,i)\subseteq h(s,i)$. Similar to [6], we further denote $Sub(A)$ the set of all deterministic sub-machines of FSM A. We also use the definition of a deterministic path in FSM A since only such paths can occur in its deterministic sub-machines. A *deterministic path* P of FSM A starts at the initial state and has no transitions with different next states and/or outputs for the same state-input combination.

As usual, function h can be extended to the set I^* of finite input sequences. Given state $s\in S$ and input sequence $\alpha=i_1 i_2...i_k\in I^*$, output sequence $o_1 o_2...o_k\in h(s,\alpha)$ if there exist states $s_1=s, s_2, \ldots, s_k, s_{k+1}$ such that $(s_{j+1},o_j)\in h(s_j,i_j)$, $j = 1,\ldots,k$. We let the set $h^o(s,\alpha) = \{\gamma \mid \exists s' \in S\ [(s',\gamma)\in h(s,\alpha)]\}$ denote the *output projection* of h, while denoting $h^s(s,\alpha) = \{s' \mid \exists \gamma \in Y^*\ [(s',\gamma)\in h(s,\alpha)]\}$ the *state projection* of h. Input/Output sequence $i_1 o_1 i_2 o_2...i_k o_k$ is called a *trace* of A if $o_1 o_2...o_k\in h^o(s_0,i_1 i_2..i_k)$. We also use the notation $h^s{}_\gamma(s,\alpha)$ to denote the set $\{s' \mid (s',\gamma)\in h(s,\alpha)]\}$ of all states where the sequence α can take FSM A from the initial state with the output response γ. For appropriate (s,γ) the set $h^s{}_\gamma(s,\alpha)$ can be empty. If the set $h^s{}_\gamma(s,\alpha)$ has the unique state s then we say the state s is *observably reachable* from the initial state via the trace α/γ.

Given states s_1 and s_2 of complete FSM A, states s_1 and s_2 are *equivalent,* written $s_1\cong s_2$, if for each input sequence $i_1 i_2...i_k\in I^*$, it holds that $h(s_1,i_1 i_2...i_k) = h(s_2,i_1 i_2...i_k)$. If states s_1 and s_2 are not equivalent then they are *distinguishable,* written $s_1 \neq s_2$. Given complete FSM A, sequence $\alpha\in I^*$ such that $h(s_1,\alpha)\neq h(s_2,\alpha)$ is said to *distinguish* states s_1 and s_2. FSM A with pair-wise distinguishable states is called a *reduced* FSM.

Complete FSMs $A = (S,I,O,h,s_0)$ and $B = (T,I,O,g,t_0)$ are *equivalent,* written $A\cong B$, if their sets of traces coincide. It is well known, given complete deterministic FSM A, there always exists a reduced FSM that is equivalent to A.

2.1 A System of Two Communicating FSMs

Many complex systems are typically specified as a collection of communicating components. We consider here a special case, where the system consists of two Communicating FSMs (*ComFSMs*), called embedded machine (or M_2) and context machine (or M_1). We let the alphabet X and Y represent the externally observable input/output actions of the system, while the U and Z alphabets represent the internal (hidden) input/output interactions between the two components. The two (deterministic) FSMs communicate asynchronously via bounded input queues where input/output messages are stored. An FSM produces an output in response to each input. We assume that the system at hand has at most one message in transit, i.e. the next external input is submitted to the system only after it produced an external output y to the previous input. Under these assumptions, the collective behavior of the two communicating FSMs can be described by *product machine.* The product machine $M_1 \times M_2$ is a Labeled Transition System (LTS) that describes the joint behavior of the component machines in terms of all actions within the system. If the product machine has a cycle labeled only with internal actions from the alphabet $U \cup Z$ then the system falls into live-lock when an input sequence leading to this cycle is applied; i.e. the system will not produce any external output. Here we accept a catastrophic interpretation of live- locks and, similar to [9], say the composed FSM of M_1 and M_2 is not defined. Otherwise, a *composed machine,* denoted as *Reference System(RS)* $= M_1 \Diamond M_2$, can be obtained from the product machine by hiding all internal actions in the product machine, and pairing input with output actions [9].

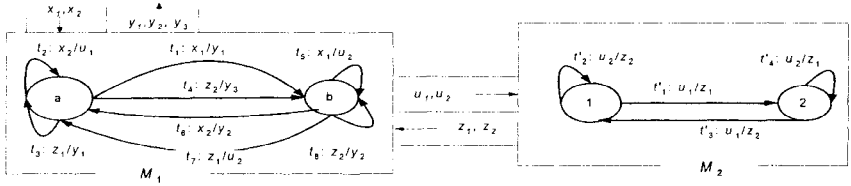

Figure 1. A system of two ComFSMs M_1 and M_2

As an example, we consider the two machines M_1 and M_2 shown in Fig. 1. The set of external inputs is $X = \{x_1, x_2\}$, the set of external outputs is $Y = \{y_1, y_2, y_3\}$, the set of internal inputs is $U = \{u_1, u_2\}$, and the set of internal outputs is $Z = \{z_1, z_2\}$. Their corresponding reference system $RS = M_1 \Diamond M_2$ is shown in Fig. 2.

A tester, implements a given test by executing external input sequences (test cases) simultaneously against both the SUT consisting of the implementation of M_1 and M_2, and the reference system in order to generate

the observed and expected outputs. If for each test case, the sequences of the observed and expected outputs coincide then the system is said to *pass* the test suite.

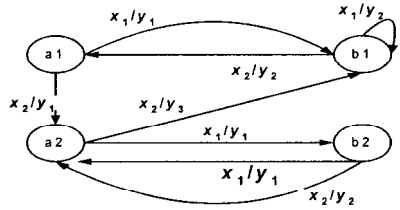

Figure 2. Reference System ($M_1 \Diamond M_2$) of the M_1 and M_2 of Fig. 1

2.2 A Fault Model for the System of Communicating FSMs

We consider a fault model based on output and transfer faults of a deterministic FSM [2]. Given complete deterministic specification and implementation FSMs $\langle S, I, O, \delta, \lambda, s_0 \rangle$ and $\langle S, I, O, \Delta, \Lambda, s_0 \rangle$, we say a transition (s,i) has an *output fault* if $\delta(s,i)=\Delta(s,i)$ while $\lambda(s,i)\neq\Lambda(s,i)$. An implementation has a *single output fault* if one and only one of its transitions has an output fault. We say that a transition has a *transfer fault* if $\delta(s,i)\neq\Delta(s,i)$, i.e. given state s and input i, the implementation enters a different state than that specified by the next-state function of the specification. An implementation has a *single transfer fault* if there is no output fault in the implementation and one and only one of its transitions has a transfer fault. We say that the implementation under test of the given system may have *multiple faults* if several transitions have output or transfer faults. Here we notice the fault model based on output and transfer faults is often used for diagnosis of a system decomposed into components, where only one component may be faulty [5]. In our context, the specification is a decomposed system; i.e. its implementation is a system of two *ComFSMs,* where at most one of these machines is faulty.

3. DIAGNOSIS PROBLEM

3.1 Diagnosis Problem Statement

Let M_1 and M_2 be complete deterministic FSMs representing the specifications of the components of the given system while M'_1 and M'_2 are their corresponding implementations. We propose an adaptive method for diagnostic test derivation. If the implementation system does not pass a given test suite, our algorithm enables to decide whether it is possible to

identify a faulty component of the given system under the assumption that multiple faults may occur only in one of the implementations M'_1 or M'_2. Furthermore, if this is possible, the algorithm enables to decide whether it is possible to locate the faulty transitions within the faulty component, and if possible it locates them. Moreover, the algorithm draws the conclusion *"Faults cannot be captured by the assumed fault model"* if it has been detected that the implementation at hand has faults that cannot be captured by the assumed fault model.

The diagnostic method can be used for the case where multiple transfer or output faults can occur in one of the implementation component FSMs M'_1 or M'_2. However, without loss of generality and for simplicity of presentation, hereafter, we assume that only output faults may occur.

3.2 An Overview of the Diagnostic Approach

Let $RS = M_1 \Diamond M_2$ be a specification system while *TS* is a conformance test suite. If the composition $M'_1 \Diamond M'_2$ of the implementations M'_1 and M'_2 of the given system produces unexpected output responses to the given test suite *TS,* then the composition $M'_1 \Diamond M'_2$ is not equivalent to *RS,* i.e. either M'_1 or M'_2 is a faulty implementation. Our objective is to determine whether M'_1 is not equivalent to M_1 while M'_2 and M_2 are equivalent, or vice versa.

In order to determine whether the output responses of *TS* can be produced when FSM M'_2 has output faults and M'_1 is equivalent to its specification, we derive the FSM $(M_2)_a^{out}$ by adding new (faulty) transitions with all possible outputs to each transition of M_2 ,and then we combine the obtained non-deterministic FSM with M_1. We call the obtained non-deterministic machine, $M_1 \Diamond (M_2)_a^{out}$, the Fault Function (FF) of the embedded component (or *FF-Embedded*). Similarly, we derive the FF of the context M_1, *FF-Context* $= (M_1)_a^{out} \Diamond M_2$, to determine whether the output responses of *TS* can be produced when FSM M'_1 has output faults and M'_2 is equivalent to its specification. Fault functions were introduced in [8] to represent in a concise way all mutants of a given FSM with a given type of implementation errors.

The set of all deterministic sub-machines of $M_1 \Diamond (M_2)_a^{out}$ (or $(M_1)_a^{out} \Diamond M_2$) includes each implementation $M_1 \Diamond M'_2$ (or $M'_1 \Diamond M_2$), where output faults are possible in the implementation of component FSM M_2 (or M_1). However, the set also includes superfluous sub-machines that do not correspond to a composition of any possible deterministic component machines. This is due to the fact that while deriving the composition, we do

not take into consideration that for a specific state-input combination (s,i), one and only one output is possible in a deterministic implementation.

Since our implementation system is deterministic we do not take into account non-deterministic paths of machines $M_1 \Diamond (M_2)_a^{out}$ and $(M_1)_a^{out} \Diamond M_2$. Moreover, we also remove from $M_1 \Diamond (M_2)_a^{out}$ and $(M_1)_a^{out} \Diamond M_2$ a behavior that does not agree with the observed outputs to the applied test suite *TS*. In Section 4, we describe the algorithm that removes from machine $M_1 \Diamond (M_2)_a^{out}$ (or $(M_1)_a^{out} \Diamond M_2$) sub-machines whose output responses to the test suite do not agree with those obtained by applying the test suite *TS* to the SUT.

If the SUT is equivalent to a deterministic sub-machine of $M_1 \Diamond (M_2)_a^{out}$ then the faulty machine is M_2, and if it is equivalent to a deterministic sub-machine of $(M_1)_a^{out} \Diamond M_2$, then the faulty machine is M_1. However, if the SUT is equivalent to a deterministic sub-machine of $M_1 \Diamond (M_2)_a^{out}$ and to a deterministic sub-machine of $(M_1)_a^{out} \Diamond M_2$, then the faulty machine cannot be identified. This is due to the fact that there are some possible faults in M_1 and some possible faults in M_2 that cause the same observable behavior of the SUT. If none of the above cases applies, we conclude that the implementation has faults that are not captured by the considered fault model.

In order to draw one of the above conclusions, we should have test cases such that by observing the output responses of the SUT to these test cases, we can distinguish the SUT and sub-machines of $(M_1)_a^{out} \Diamond M_2$ and of $M_1 \Diamond (M_2)_a^{out}$. If the machines $(M_1)_a^{out} \Diamond M_2$ and $M_1 \Diamond (M_2)_a^{out}$ have no equivalent deterministic sub-machines then there exists a so-called *distinguishing set* of input sequences [6] such that given the set of deterministic output responses to these input sequences, we always can determine whether the machine under test is a sub-machine of $(M_1)_a^{out} \Diamond M_2$ or $M_1 \Diamond (M_2)_a^{out}$. The algorithm for deriving such a distinguishing set is proposed in [6]. We illustrate this algorithm in Section 4. In other cases, it may happen that there are sub-machines of $(M_1)_a^{out} \Diamond M_2$ and others of $(M_1)_a^{out} \Diamond M_2$ that are equivalent, but these sub-machines are not equivalent to the SUT. In this case, the observed outputs to the given test suite are insufficient to distinguish between these sub-machines and the SUT. Therefore, more test cases must be generated and added to the test suite. This can be done by breaking *FF-Embedded* and *FF-Context* into sub-machines, and by comparing each pair of the obtained sub-machines. Each time when two obtained sub-machines become distinguishable, their distinguishing test is added to the test suite. This enables the elimination of all sub-machines whose output responses to a distinguishing test are different from those observed by applying this test to the SUT. We break the machines by fixing

some transitions as deterministic transitions, i.e. by reducing the non-determinism of Fault Functions. In the worst case, we can come up with explicit enumeration of all faulty machines. However, as the considered examples show, we usually need to fix only a small number of transitions in order to draw a conclusion.

The details of the diagnostic method are presented in Section 5. Meanwhile, in the following section we present its constituent algorithms illustrated by a working example.

3.3 Working Example

Consider the two machines M_1 and M_2 given in Fig. 1, and their corresponding reference system $RS = M_1 \Diamond M_2$ shown in Fig. 2. A given complete test suite TS derived from RS is $TS = \{x_1x_1x_1, x_1x_2x_2, x_2x_1x_1x_2, x_2x_1x_2x_2, x_2x_2x_1\}$. TS is derived using the method presented in [1], and it detects any complete FSM $M'_1 \Diamond M'_2$ that is not equivalent to RS, under the assumption that FSMs M'_1 and M'_2 have up to two states. The set of expected output responses to the TS is as follows:
$\lambda(s_0, x_1x_1x_1) = y_1y_2y_2$; $\lambda(s0, x_1x_2x_2) = y_1y_2y_1$; $\lambda(s_0, x_2x_1x_1x_2) = y_1y_1y_1y_3$; $\lambda(s0, x_2x_1x_2x_2) = y_1y_1y_2y_3$; $\lambda(s0, x_2x_2x_1) = y_1y_3y_2$.

Let us assume that the composition $M'_1 \Diamond M'_2$ of the implementation component FSMs M'_1 and M'_2 produces unexpected output responses
$y_1y_3y_3$ to $x_1x_1x_1$; $y_1y_1y_3y_2$ to $x_2x_1x_1x_2$ and $y_1y_3y_3$ to $x_2x_2x_1$ **(1)**

Thus, the composition $M'_1 \Diamond M'_2$ is not equivalent to RS, i.e. either M'_1 or M'_2 is a faulty component implementation.

Figure 3 includes the *FF-Embedded* machine, $M_1 \Diamond (M_2)_a^{out}$, and the *FF-Context* $(M_1)_a^{out} \Diamond M_2$, obtained as described in Section 3.2. In this example, the *FF-Context* is derived under an assumption that in the faulty context implementation, external outputs can only be replaced with external outputs and internal outputs can only be replaced with internal outputs, respectively. This is done in order to have a simple and more readable working example.

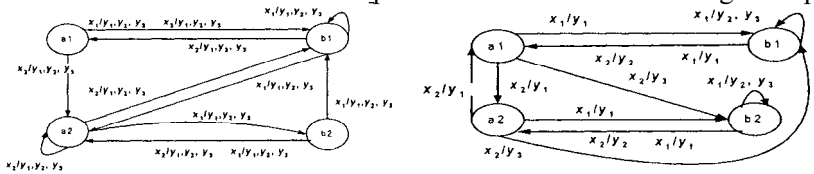

Figure 3. (a) Fault function of Embedded Component (b) Fault Function of Context

4. DISTINGUISHING NON-DETERMINISTIC FSMS

In this section, we present the different algorithms of the test suite derivation method, given in the subsequent section, for locating a faulty component FSM.

4.1 Removing Sub-machines of a Non-Deterministic FSM

The following algorithm is used to remove from $(M_1)_a^{out} \Diamond M_2$ (and from $M_1 \Diamond (M_2)_a^{out}$) sub-machines whose output responses to the test cases of *TS* disagree with those obtained by applying these test cases to the SUT.

Given a complete non-deterministic FSM $A = (S, I, O, h, s_0)$ and a set V of deterministic sequences over alphabet $(IO)^*$, the algorithm returns a smallest sub-machine A^r of A which has the property that each sub-machine of A that comprises V as a subset of its traces is a sub-machine of A^r. We note that in our case, the input parts of the sequences of the initial set V are those of the given test suite *TS,* and the output parts are those observed by applying *TS* to the SUT.

Algorithm 4.1. Removing from FSM A sub-machines that do not match V

Input: A complete non-deterministic FSM $A = (S, I, O, h, s_0)$ and a set V of deterministic sequences over alphabet $(IO)^*$

Output: The smallest subFSM $A^r = (S, I, O, h, s_0)$ of A that contains each sub-machine of *A,* such that the set of its traces comprises *V,* if exists.

Step-1. Given FSM $A_1 = A$ and the set V of sequences over alphabet $(IO)^*$, we derive the tree $Tree_1$ of all deterministic paths through A_1 labeled with sequences of *V.* Assign i:=1 and go-to Step-2.

Step-2. If there exists a sequence in V such that no path in $Tree_i$ is labeled with this sequence, then there is no sub-machine in A such that V is a subset of the set of traces of that sub-machine (A^r does not exist, end of Algorithm 4.1). Otherwise, we build a machine A_{i+1} which is a sub-machine of A_i as follows. For each observably reachable node in $Tree_i$, we copy into the corresponding state in A_{i+1} the outgoing transitions from this node. If there are several observably reachable nodes in $Tree_i$ with the same label, we copy for the corresponding state in A_{i+1} only the matching transitions at these nodes (same input/output/next-state values). If there are no matching transitions in $Tree_i$, then there is no sub-machine in A such that V is a subset of the set of traces of that sub-machine (A^r does not exist, end of Algorithm 4.1). For each state in A_i, that labels only non-observably reachable nodes in $Tree_i$, we copy all the outgoing transitions from machine A_i into A_{i+1}. If no

transition in the machine A_i has been changed (i.e. $A_{i+1} = A_i$), then we have $A^r = A_i$ (End of Algorithm 4.1). Otherwise, go-to Step-3.

Step-3. At this step we trim $Tree_i$ in order to obtain $Tree_{i+1}$ using the machine A_{i+1} obtained by Step-2 above. For each path in $Tree_i$ that has a node where the output and/or next node of the transition do not match machine A_{i+1}, remove this transition and its sub-tree. If all outgoing transitions from some node for an appropriate input have been removed, we remove the incoming transition to this node. If all transitions from the root node for an appropriate input have been removed, then there is no sub-machine in A such that V is a subset of the set of traces of that submachine (A^r does not exist, end of Algorithm 4.1). If the trees $Tree_i$ and $Tree_{i+1}$ coincide, then the machine $A^r = A_{i+1}$ is derived (End of Algorithm 4.1). Otherwise, increment i by 1 and go back to Step-2.

As an example, we consider $Tree_1$ in Fig. 4-a generated for the fault function of the context, i.e. $A_1 = (M_1)_a^{out} \Diamond M_2$ in Fig. 3-b, using Step-1 and *TS*. We do not include in $Tree_1$ the paths that do not match the observed output of *TS*. For example, for all test cases in the *TS* that start with the input x_2, the observed output for x_2 is y_1. Therefore, the paths of $(M_1)_a^{out} \Diamond M_2$ that start from the initial state a_1 by a transition labeled with a label other than x_2/y_1 are not included in $Tree_1$. Moreover, we do not include in $Tree_1$ any non-deterministic path. For example, we do not include the path $a_1 \xrightarrow{x_1/y_1} b_1 \xrightarrow{x_1/y_3} b_1 \xrightarrow{x_1/y_3} a_1$.

In $Tree_1$, the root node a_1 is observably reachable through the empty sequence. Therefore, in Step-2, we copy the outgoing transitions of that node into A_2 (shown in Fig. 4-b), i.e. transitions, $a_1 \xrightarrow{x_1/y_1} b_1$, $a_1 \xrightarrow{x_2/y_1} a_2$ and $a_1 \xrightarrow{x_2/y_1} b_1$. Moreover, in $Tree_1$, starting from the root node a_1, the node b_1 is observably reachable through the sequence x_1/y_1. Therefore, in Step-2, we copy the outgoing transitions of b_1 from $Tree_1$ into A_2. Nodes a_2 and b_2 in $Tree_1$ are only non-observably reachable. Therefore, we copy from A_1 in Fig. 3-b the outgoing transitions from states a_2 and b_2 into A_2.

Afterwards, using A_2 in Step-3, we consider in $Tree_1$, starting from the root node a_1, the outgoing transitions from the node b_1 that is reached through the sequence x_2/y_1. We notice that all its outgoing transitions do not match A_2. Therefore we trim the sub-tree of this node, and since all outgoing transitions for inputs x_1 and x_2 from this node are removed, we remove its

incoming edge, and we get $Tree_2$, which is equal to $Tree_1$ except that the shaded area TRIM-1 is removed.

Back to Step-2, by considering $Tree_2$, the root node a_1 is observably reachable through the empty sequence. Moreover, starting from the root node a_1, the ending nodes a_1, b_1, and a_2 are observably reachable through the sequences $x_1/y_1x_2/y_2$, x_1/y_1, and x_2/y_1, respectively, and node b_2 is observably reachable through the sequence $x_2/y_1x_1/y_1$. Therefore, for these nodes, we copy their matching outgoing transitions from $Tree_2$ into A_3 in Fig.5-a.

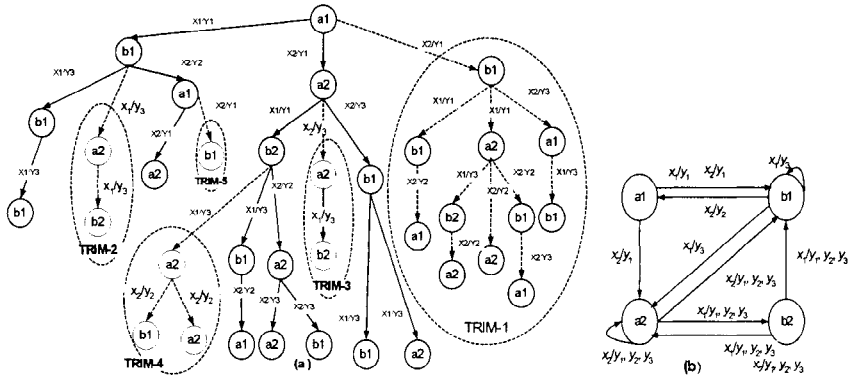

Figure 4 . (a) $Tree_1$ obtained by removing non-deterministic paths from $(M_1)_a^{out} \Diamond M_2$ of Fig.3-b , (b) Machine A_2 that correspond to $Tree_1$ depicted in Fig. 4-a

Using the same reasoning we trim from $Tree_2$ the shaded areas of TRIM-3, TRIM-4, and TRIM-5 and we obtain $Tree_3$.

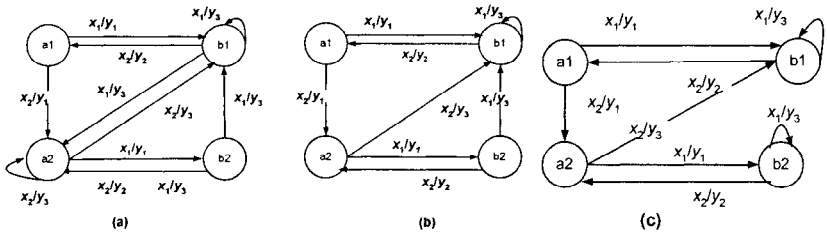

Figure 5. (a) Machine A_3 that correspond to $Tree_2$, (b) Machine A_4 that correspond to $Tree_3$, (c) Machine for the embedded component

Back to Step-2, by considering $Tree_3$, we notice that all nodes are observably reachable. We copy all the matching outgoing transitions of these nodes and obtain the final machine A_4 shown in Fig. 5-b.

We apply Algorithm 4.1. to the *FF-Embedded,* i.e. for $M_1 \Diamond (M_2)_a^{out}$ of Fig. 3-a, and we obtain the machine shown in Fig. 5-c.

Theorem 4.1. Given a complete non-deterministic FSM $A = (S, I, O, h, s_0)$ and a set V of deterministic sequences over alphabet $(IO)^*$, if there exist a sub-machine B of A that has V as a subset of its traces, then the FSM A^r is

derived by Algorithm 4.1 and it includes B. Otherwise, the FSM A^r does not exist.

Proof: In Step-2 of Algorithm 4.1, if there exists no deterministic path in $Tree_i$ labeled with some sequence $\alpha/\beta \in V$, then there is no deterministic sub-machine of A that can produce the observed output sequence β to the input sequence α, i.e. there is no sub-machine A^r in A such that V is a subset of traces of A^r. Moreover, In Step-3, if all the transitions from the root node in $Tree_i$ have been removed, then there is no deterministic sub-machine A^r in A such that V is subset of traces of A^r. The transitions of A are changed only during Step-2 for the states that label observably reachable nodes of the corresponding tree. For this reason, it is enough to show that each sub-machine B of A_i that has V as a subset of its traces, is a sub-machine of A_i+1. Let state s label an observably reachable node of $Tree_i$ that is reachable through some sequence $\alpha/\beta \in V$. Any sub-machine of A_i that has V as a subset of its traces must reach state s after applying the sequence α. Therefore, all transitions from the state s of B match transitions from this node, i.e. B is a sub-machine of A_i+1.

4.2 Distinguishing the Sets of Deterministic Sub-Machines of Two Non-Deterministic FSMs

If the two sub-machines of the FSMs *FF-Context* and *FF-Embedded* obtained after using the above procedure for removing behaviors that do not match observed outputs, do not have equivalent sub-machines then we can derive a distinguishing set Dis_{Set} that allows us to recognize which of the component FSMs is faulty. This set can be constructed using the algorithm given in [6]. In other words, let machines $M_1 \Diamond (M_2)_a^{out}$ and $(M_1)_a^{out} \Diamond M_2$ be distinguished with the distinguishing set Dis_{Set}. Let also P be a sub-machine of $M_1 \Diamond (M_2)_a^{out}$ or of $(M_1)_a^{out} \Diamond M_2$. Then by observing the output responses of P to sequences in the set Dis_{Set}, we can always conclude whether $P \in Sub(M_1 \Diamond (M_2)_a^{out})$ or $P \in Sub((M_1)_a^{out} \Diamond M_2)$, i.e. the diagnostic problem is always solvable. Therefore, we derive, using the algorithm given in [6], a distinguishing set Dis_{Set} for the FSMs *FF-Context* and *FF-Embedded* in order to recognize a sub-machine that corresponds to the faulty SUT.

As an example, we consider the machines *FF-Context* (machine $A4$ of Fig. 5-b) and *FF-Embedded* (Fig. 5-c) obtained after deleting sub-machines with traces that do not match observed output responses. The sequence $x_2x_1x_1x_2x_2$ distinguishes these machines. Therefore, if we apply the input x_2 after the input sequence $x_2x_1x_1x_2$ and the implementation at hand produces the output response y_1 to the tail input x_2, then we conclude that M'_1 is the

faulty implementation. If the output y_3 is produced to the tail input x_2, then we conclude that M'_2 is faulty. If the implementation produces the output different from y_3 and y_1, then the implementation at hand has faults that cannot be captured by the assumed fault model.

4.3 Determining a Superfluous Sub-machine

Due to the considered fault model, the SUT M is a sub-machine of $M_1 \Diamond (M_2)_a^{out}$ or $(M_1)_a^{out} \Diamond M_2$. However, we mentioned above that not each sub-machine of $M_1 \Diamond (M_2)_a^{out}$ (or $(M_1)_a^{out} \Diamond M_2$) can be obtained through output faults in the implementation of M_2 (or M_1). The reason is that we do not take into account deterministic and non-deterministic paths when combining the compact representation $(M_2)_a^{out}$ of all possible implementations of M_2 with M_1 (or the compact representation $(M_1)_a^{out}$ of all possible implementations of M_1 with M_2), and thus we may obtain superfluous sub-machines. Therefore, it may happen that the SUT M is equivalent to a sub-machine of $M_1 \Diamond (M_2)_a^{out}$ and to a submachine of $(M_1)_a^{out} \Diamond M_2$, but there is no sub-machine $M'_2 \in (M_2)_a^{out}$ such that $M_1 \Diamond M'_2$ is equivalent to M. In this case, only the implementation of M_1 is faulty. Thus, we have the following problem.

Given FSM $M_1 \Diamond (M_2)_a^{out}$ and its sub-machine M, we must check whether there exists FSM $M'_2 \in (M_2)_a^{out}$ such that $M = M_1 \Diamond M'_2$. To solve the problem we can project sub-machine M onto the set of states of M_2 and input and output alphabets of M_2. There exists FSM $M'_2 \in (M_2)_a^{out}$ such that $M = M_1 \Diamond M'_2$ if and only if the obtained FSM is deterministic. A sub-machine M of $M_1 \Diamond (M_2)_a^{out}$ for which there is no $M'_2 \in (M_2)_a^{out}$ such that FSMs $M_1 \Diamond M'_2$ and M are equivalent, is called a *superfluous* sub-machine.

5. FAULT DIAGNOSIS ALGORITHM

Algorithm 5.1. Recognizing a faulty component FSM

Input: Composition $M \cong M_1 \Diamond M_2$ of two FSMs M_1 and M_2, and the set $V = TS$ of sequences over alphabet $(IO)^*$,

Output: Verdict *"Component FSM* M_1 (or M_2) *is faulty",* or verdict *"Both* M_1/M_2 *could be faulty"* when there is a possible faulty implementation of M_1 and a possible faulty implementation of M_2 that cause the same observable behavior as the implementation at hand, or

verdict *"Faults cannot be captured by the assumed fault model"* if it has been detected that the implementation at hand has faults that can not be captured by the assumed fault model.

Step-1. Derive machines, $A_1 = (M1)_a^{out} \lozenge M_2$ (Fault Function of M_1) and $A_2 = M_1 \lozenge (M_2)_a^{out}$ (Fault Function of M_2). Let the set R_1 be equal to $\{A_1\}$, and the set R_2 be equal to $\{A_2\}$.

Step-2. For each machine say A_k in the sets R_1 and R_2, call Algorithm 4.1. to obtain the smallest sub-machine A^r of A_k which includes all sub-machines of A_k that have V as a subset of their traces. If such an A^r exist, replace A_k by A^r. Otherwise, remove A_k from the corresponding set R_1 or R_2.

If the sets R_1 and R_2 are empty, then the implementation at hand has a fault that is not captured by the assumed fault model (End of diagnosis algorithm).
If the set R_1 (or R_2) is empty, then we conclude the other machine M_2 (or M_1) is faulty (End of diagnosis algorithm).
Otherwise, check, as described in [6], whether there are two machines, say A_i in R_1 and A_j in R_2, that are distinguishable.

♦- If there exist such two machines, we obtain, using the algorithm given in [6], the $Dist_{set}$ that distinguish them, and we apply the input sequences of this set to the SUT.

--If $|R_1| = 1$ and $|R_2| = 1$, i.e. $R_1 = \{A_i\}$ and $R_2 = \{A_j\}$, then:

-If the output responses of the SUT to the $Dist_{set}$ are different from those expected by both machines A_i and A_j, then we conclude that the implementation at hand has faults that cannot be captured by the assumed fault model (End of diagnosis algorithm).

-Else, if the output responses of the SUT are different than those expected by A_j (or A_i), then we conclude that M_1 (or M_2) is the faulty machine (End of diagnosis Algorithm).

--If $|R_1| > 1$ or $|R_2| > 1$, then after observing the output responses of the SUT to the sequences in $Dist_{set}$, we remove A_i (or A_j) from the set R_1 (or R_2) if these responses are different than those expected by A_i (or A_j). Then, we add the sequences in $Dist_{set}$ with the observed output responses to V, and we return return back to Step-2.

♦- If all the machines in R_1 are indistinguishable from those in R_2, then

--If R_1 and R_2 have only deterministic machines, then check whether all the machines in R_1 and in R_2 are superfluous as described above.

-If all the sub-machines in R_1 (or in R_2) are superfluous, then machine M_2 (or M_1) is faulty (End of diagnosis algorithm).

-Else, If there exist a sub-machine in R_1 and another in R_2 that are not superfluous, then we conclude that "Both_M_1/M_2 could _be_ faulty". There is a possible faulty implementation of M_1 and a possible faulty implementation of M_2 that cause the same observable behavior as that of the implementation at hand

--Else, if the set R_1 (or R_2) has at least one non-deterministic FSM, then we break that machine into k machines by fixing one of its non-deterministic transitions. Then, we replace that machine in the set R_1 (or R_2) with the obtained sub-machines, and we go back to Step-2.

As another example, suppose that the implementation of the composed system $M'_1 \lozenge M'_2$ of specifications in Fig. 1 produces the unexpected output responses $y_1 y_1 y_2 y_2$ to the input sequence $x_2 x_1 x_1 x_2$ of *TS*. This can happen if M'_1 has an output fault; namely it produces y_2 instead of u_2 on executing transition t_5.

By applying the diagnostic method described above, we find that M'_1 can not be identified as the faulty implementation since there exists a faulty implementation M'_2 of M_2 such that $M_1 \lozenge M'_2$ and $M'_1 \lozenge M_2$ are equivalent. It is the case when M'_2 produces z_2 instead of z_1 on executing t'_4.

6. FURTHER RESEARCH WORK

In this paper we presented a method for diagnostic test derivation when the system specification and implementation are given in the form of two communicating finite state machines and at most a single component machine can be faulty. The algorithm can be extended for locating the faults within the faulty machine (if possible). The idea here is to locate the faulty implementation that corresponds to the faulty machine. This implementation is a sub-machine of $(M_1)_a^{out} \lozenge M_2$ when M_1 has a faulty implementation, or a sub-machine of $M_1 \lozenge (M_2)_a^{out}$ when M_2 has a faulty implementation. Therefore, the algorithm can be extended to decide which sub-machine of $(M_1)_a^{out} \lozenge M_2$ (or $M_1 \lozenge (M_2)_a^{out}$) is equivalent to the system at hand, i.e. comparison between sub-machines of a given FSM can be added to the algorithm. We note that sometimes it is not possible to locate the faulty sub-machine. It is the case when the SUT has the same observable behavior for different output faults of the faulty machine. Moreover, the algorithm can be extended for locating faults when the specification and implementation systems are given in the form of a single FSM

Currently, we are performing experiments to estimate the effectiveness and the complexity of the method. The preliminary results obtained for the case when only single output faults in a component are taken into account, show that almost always the faulty component can be identified.

REFERENCES

[1] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou and A. Ghedamsi.'Test selection based on finite state models', *IEEE Trans. SE*-17, No. 6, 199 1, pp. 591-603.

[2] A. Ghedamsi and G. v. Bochmann.'Test result analysis and diagnostics for finite state machines', *Proc. of the 12-th ICDS,* Yokohama, Japan, 1992.

[3] A. Ghedamsi, G. v. Bochmann and R. Dssouli.'Diagnostic Tests for Communicating Finite State Machines', *Proc. of the 12th IEEE Internaitonal Phoenix Conference on Communications,* Scottsdale, USA, March 1993.

[4] A. Ghedamsi, G. v. Bochmann and R. Dssouli. 'Multiple fault diagnosis for finite state machines', *Proc. of IEEE INFOCOM'93,* 1993, pp.782-791.

[5] J. de. Kleer and B.C. Williams.'Diagnosing multiple faults', *Artificial Intelligence* 32(1), 1987, pp. 97-130.

[6] I. Koufareva. 'Using non-deterministic FSMs for test suite derivation', *Ph.D. Thesis,* Tomsk State University, Russia, 2000 (In Russian).

[7] D. Lee and K. Sabnani. 'Reverse engineering of communication protocols', *Proc. of ICNP,* October 1993, pp. 208-216.

[8] A. Petrenko and N. Yevtushenko.'Test suite generation for a FSM with a given type of implementation errors', *Proc. of the 12th IWPSTV,* 1992, pp. 229-243.

[9] A. Petrenko, N. Yevtushenko, G. v. Bochmann, and R. Dssouli. 'Testing in context: Framework and test derivation', *Computer Communications Journal,* Special issue on protocol engineering, 1996, pp. 1236-1249.