Diagnostic Tests for a Class of Non-Deterministic Finite State Machines

A. Ghedamsi, G. v. Bochmann, R. Dssouli and G. Luo

Université de Montréal
Département d'Informatique et de Recherche Opérationnelle
C.P.6128, Succ. "A", Montréal, Canada, H3C 3J7
Tel: (514) 343-6111 (3508), Fax: (514) 343-5834

Abstract

We propose a generalized diagnostic algorithm for the case where a software system specification (implementation) is given in the form of an observably non-deterministic finite state machines (ONFSM). Such an algorithm localizes the faulty transition in the system once the fault has been detected. It generates, if necessary, additional diagnostic test cases which depend on the observed symptoms and which permit the location of the detected fault. The algorithm guarantees the correct diagnosis of any single (output or transfer) fault in a software system represented by an ONFSM. A simple example is used to demonstrate the functioning of the different steps of the proposed diagnostic algorithm.

***Key words*:** Diagnostic, Test, Observably non-deterministic finite state machine, Symptom, Conflict set, Candidate.

Diagnostic Tests for a Class of Non-Deterministic Finite State Machines

1. Introduction

Testing is an important step in the development cycle of any system (i.e. software, communication protocol or hardware). A lot of research work has been directed towards such tests [Fuji 91, More 90, Davi 88, Sabn 88, Ural 87, Nait 81, Chow 78, Gone 70]. At the same time, in the software domain where a software system may be represented by an FSM model, very little work has been done for diagnostic and fault localization problems [Ghed 92, Koka 90, Kore 88]. Diagnostic is a well documented subject in other areas, such as Artificial Intelligence (AI), complex mechanical systems and medicine [Scho 76]. Therefore, most of the concepts and terms used in this paper are imported from those domains.

In model-based diagnostics [Klee 87, Reit 87], we assume the availability of the physical system (i.e, implementation) which can be observed, and its model (i.e, specification) from which predictions can be made about its behavior. It is necessary to know how the system or the machine under test is supposed to work in order to be able to know why it is not working correctly.

Often the specification of a model-based system is described in a structured manner. Therefore, a system is seen as a set of components connected to each other in a specific way. A **component** is seen as one of many smaller sub-systems in a larger system. The behavior of the larger system is, therefore, described in terms of its components behaviors. The **structure** (organization) of a system can be defined as a relationship (i.e, physical connection, procedure call,...) between the different components of the system. In order to diagnose this kind of systems, models and their corresponding real systems are assumed to have the same components and the same structure. **Observations** of inputs and outputs show how the system is behaving, while **expectations** tell us how it is supposed to behave. The differences between expectations and observations, which are called **"symptoms"**, hint the existence of one or several differences between the model and its system. In order to explain the observed symptoms, a diagnostic process should be initiated. It consists mainly of performing the following two tasks: the generation of candidates and the discrimination between candidates [Klee 87].

Task 1: Generation of candidates: This process uses the identified symptoms and the model to deduce some diagnostic candidates. Each **diagnostic candidate** is defined to be the minimal difference, between the model and its system, capable of explaining all symptoms. It indicates the failure of one or several components in the system.

Task 2: Discrimination between candidates: Once the step of candidate generation terminates, we often end up with a huge number of diagnostic candidates. To reduce their number, two main techniques are used. The first one consists of the selection of some additional new tests called **"distinguishing tests"** [Gene 84]. The second technique consists of introducing new observation points in the implementation under investigation and executing the same tests again.

We recall that in general, the diagnostic process is a very complicated task, specially for diagnosing complicated systems. This complexity makes the achievement of the candidate generation and discrimination tasks harder. In order to solve this problem, the use of fault models is necessary (see for instance [Boch 91]). Given the system description, corresponding fault models may be established using its different levels of abstraction. Some of these fault models give all possible failures of each component in the system. They help to ease the diagnostic procedure, specially by reducing the number of the different cases which have to be considered, and hence, in reducing the number of diagnoses to be generated. It is important to note that different fault models may be used during both tasks of the diagnostic process. In the simplest case and for high level abstractions, the following fault model, based on the system decomposition into components and connections, may apply during the candidate generation phase. Each component may either be faulty or operating correctly [Klee 87]. On the other hand, and for lower level abstractions (i.e. gates or transitions levels), different uses of precise and more concrete fault models, are recorded in different areas such as the diagnostics of hardware circuits (i.e, stuck at 0/1 fault models) [Stru 89, Klee 89]. These fault models may be used during the phase of discrimination between candidates. In the software area and more precisely for FSMs, another simple fault model, based on transfer and output faults of state transitions, can be used for diagnosing software systems modelled by FSMs [Chow 78, Vuon 90, Ghed 92]. The same fault model is also used for the diagnostic approach presented in this paper.

In [Ghed 92], we introduced a single fault diagnostic algorithm for systems represented by deterministic FSMs. In this paper, we extend the applicability of our ideas to a class of non-deterministic systems. Such an extension is reached through the generalization of the diagnostic algorithm to the case where systems implementations and their models are represented by observably non-deterministic finite state machines (ONFSM) (see Definition 2). The proposed algorithm has the ability of localizing a fault once it is detected by one or several test cases possibly generated by one of the existing test selection methods.

The remainder of the paper is organized as follows. In Section 2, the observably non-deterministic finite state machine (ONFSM) model and a corresponding fault model are introduced. Section 3 includes all the details of an approach for the diagnostic of system implementations represented by the ONFSM model . In Section 4, an application example explaining the steps of the proposed diagnostic algorithm is provided. Section 5, finally, contains a concluding discussion and points for future research.

2. Non-deterministic finite state machines

2.1 The non-deterministic finite state machines model

Definition 1: A non-deterministic FSM M is defined as a quadruple (S, I, Y, Trans) where:

S : Set of states of M. It includes an initial state s_0,
I : Set of input symbols. It includes the reset input (r),
Y : Set of output symbols. It includes the null output (ε),
Trans: The transitions of M, which is a relation between present state and input on the one hand, and next state and output on the other hand:

Trans⁄(S x I) x (Y x S)

From the above definition, it is easily seen that for a single present state and input pair, the relation Trans might have different corresponding pairs of outputs and next states. Couples ((s, a), (b, s')) □ Trans are called **transitions** of the machine M. The notation **s-a/b->s'** is also used to represent a transition. For each state in the machine, a **reset transition** is used to take the machine to its initial state. It takes the symbol **r** as input and generates the symbol **ε** as output.

Definition 2: An FSM is observably non-deterministic (ONFSM) [Cern 92] if and only if for any given two transitions with equal input symbols and equal present states, if their corresponding next states are different, then their corresponding output symbols must also be different. I.e, if s-x/y'->s' and s-x/y"->s" are two transitions in an ONFSM M, and s' □ s", then y' □ y".

Definition 2 defines a special class of non-deterministic FSMs. It consists of those machines where all transitions with the same starting state have different (input/output) labels. For the rest of the paper, we will deal only with machines in this class. We also assume that the ONFSMs are completely specified and do not include internal transitions. Therefore, for every input symbol i □ I, every state in the ONFSM has at least one transition with that input. Finally and in order to deal with null outputs (i.e. ε), we assume that the output ε is determined by the application of an input and the non-observation of any output during a predetermined lapse of time. After deducing that a null output has occurred, the next input is allowed to be applied.

A graphic representation of an ONFSM example, in the form of a **State transition diagram**, is given in Figure 1.

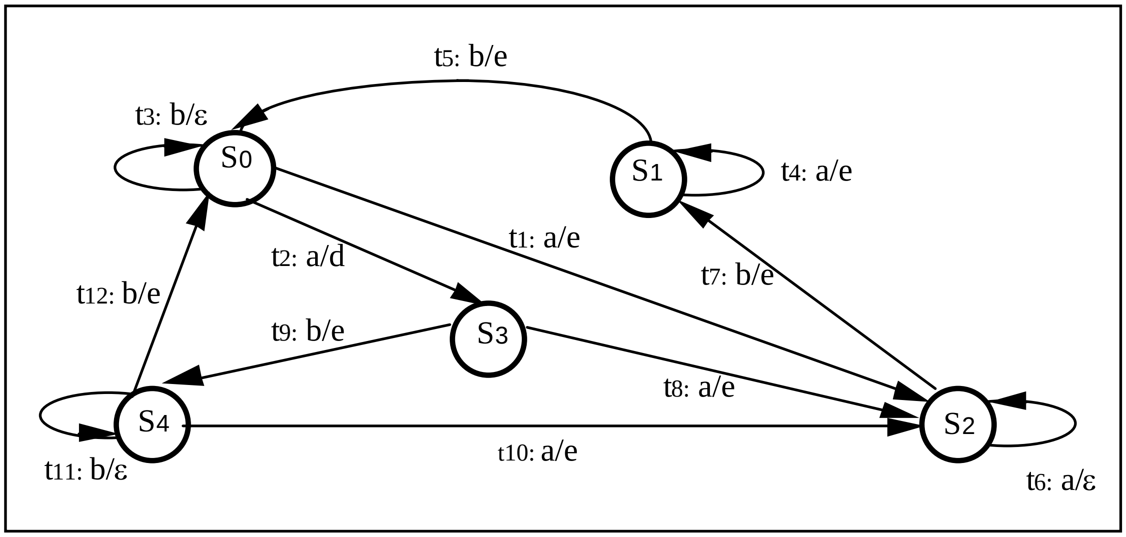


Figure 1: A state transition diagram of an ONFSM

2.2 The ONFSM fault model

The ONFSM fault model is based on errors and faults made on labeled transitions. Some of these faults, which are essential for the ONFSM-based diagnostic approach discussed in Section 3, are defined as follows:

Definition 3: Output fault: We say that a transition has an output fault if, once executed, the implementation provides an output different from the one specified by the output function.

An implementation has a **single output fault** if, one and only one of its transitions has an output fault.

Definition 4: Transfer fault: We say that a transition has a transfer fault if, once executed, the implementation enters a different state than specified by the Next-state function.

An implementation has a **single transfer fault** if, one and only one of its transitions has a transfer fault.

For our diagnostic approach presented in the following section, we assume the following fault model: the IUT may have a single output fault or a single transfer fault in one of its transitions. In other words, both the specification and the implementation are assumed to be the same with the possibility of having one difference (i.e, the output or the reached state) in at most one of their transitions.

3. The diagnostic approach

The following algorithm consists of diagnosing (with respect to its specification ONFSM) an IUT ONFSM for possible faulty transitions. Its main purpose is to identify the faulty transition and to determine the type of its fault (i.e. output or transfer). This work is mainly executed within Step 5 and Step 6 of the following algorithm. In particular, Step 5 might end up with different diagnostic candidates. In such a case and in Step 6, additional diagnostic tests should be selected in Step 6 in order to be able to isolate the faulty transition and more precisely to know to which state (in case of a transfer fault) that transition has transferred.

ALGORITHM:

Step 1: Generation of expected outputs
We assume that **a test suite "TS"** is given; it may have been obtained by one of the existing test selection methods [Luog 92]. The test suite consists of a number of test cases which are sequences of input symbols. We write TS = { tc_1, ..., tc_p}, where each **tc_i is a test case**.

Since we are dealing with non-deterministic machines, a set of valid sequences of outputs is expected for each test case in the given test suite. Such a set covers all paths (sequences of transitions) which might be executed once the corresponding sequence of input symbols is applied. Therefore, if a test case tc_i consists of m_i inputs $i_{i,1}i_{i,2}...i_{i,mi}$, the corresponding set of sequences of possible expected outputs is written as $O_i = \{ o^1_i, ..., o^q_i\}$, where $o^k_i = o^k_{i,1}o^k_{i,2}...o^k_{i,mi}$ and $o^k_{i,j}$ (k = 1, ..., q) is expected after input $i_{i,j}$.

Step 2: Generation of observed outputs

In order to be able to observe output sequences that may be produced by a given implementation

for a given input test case, we assume that the IUT satisfies a **fairness property**. Such a property states that if a test case is applied repeatedly often enough, all corresponding implementation paths (sequences of transitions) will be executed. Therefore, for each test case tc_i, we assume that a corresponding complete set of sequences of outputs, **"set of observed outputs"**, is generated by the IUT. It is written as: $\hat{O}_i = \{ \hat{o}^1_i, ..., \hat{o}^r_i\}$, where $\hat{o}^k_i = \hat{o}^k_{i,1}\hat{o}^k_{i,2}...\hat{o}^k_{i,mi}$ (k = 1, ..., r).

Step 3: Generation of symptoms

Compare outputs in the observed sets of output sequences with the corresponding sets of expected output sequences and identify all symptoms. Any difference $(O_i \neq \hat{O}_i)$ represents a **symptom**.

Definition 5: The transition $T^k_{i,j}$ of the specification where the symptom $(O_i \neq \hat{O}_i)$ (more precisely $(o^k_{i,j} \neq \hat{o}^k_{i,j})$) has been observed, is called a **symptom transition**. The faulty output corresponding to a symptom is called a **symptom output**. If we have the same symptom transition for all symptoms, that transition is called the **unique symptom transition (ust)**. The observed output generated by the ust, is called the **unique symptom output (uso)**.

Note: In order to continue the diagnostic process, different approaches might be used depending on whether the single or the multiple fault hypothesis is made. In the following, we make the assumption that **the IUT has a single fault, either output or transfer.**

Step 4: Generation of conflict sets

Algorithm: For each symptom $(O_i \neq \hat{O}_i)$, determine its corresponding conflict set. A **conflict set** for a given symptom is defined to be the set of transitions which are supposed to participate (through their execution) in the generation of the symptom; therefore, one of these transitions must be faulty. The occurrence of a fault in one of the implementation transitions might change the observed set of output sequences, which corresponds to a given test case. As a result, new output sequences might be observed, while other expected sequences might be missed. In the following, we present a new algorithm for the conflict set generation, where we consider the different possible inclusion relations the sets O_i and $\hat{O}_i$ might have.

```
If ($Ô_i$ 1 $O_i$) then
        $Plus_i = O_i - Ô_i$
        $Checkset_i = Ô_i$
        ComputeConf($Conf_i$, $Plus_i$, $Checkset_i$, $O_i$)
Else    If ($O_i$ 1 $Ô_i$) then
                $Plus_i = Ô_i - O_i$
                $Checkset_i = O_i$
                ComputeConf($Conf_i$, $Plus_i$, $Checkset_i$, $O_i$)
        Else
                $Plus_i = O_i - Ô_i$
                $Checkset_i = Ô_i - O_i$
                ComputeConf($Conf_i$, $Plus_i$, $Checkset_i$, $O_i$)
```

```
Procedure ComputeConf($Conf_i$, $Plus_i$, $Checkset_i$, $O_i$)
  $Conf_i$ = □
    Repeat
        Forall σ[ $Plus_i$ do                                   {i.e, $σ = x_1x_2x_3....x_n$ }
          R := ´
          Forall σ' [ $Checkset_i$ do                           {i.e, $σ' = x'_1x'_2x'_3....x'_n$ }
            If ( R < (Common (σ, σ'))) then                     {is R is subsequence of Common (σ, σ') ?}
                                                                {i.e, if $σ = σ_1.x_{h+1}.σ_2$ and $σ' = σ_1.x'_{h+1}.σ_3$,
                                                                where $x_{h+1}$ □$x'_{h+1}$, then Common (σ, σ') = $σ_1$ }
                R := Common (σ, σ')                             {R is affected the largest common subsequence
                $σ_f$ := σ'                                     of σ and any other sequence σ' in $Checkset_i$}
          Endforall                                             {i.e, R = $y_1y_2y_3....y_m$ , where
                                                                $m < n$, $y_i = x_i = x'_i$ for i = 1, 2, ..., m,
          R := R.$y_{m+1}$                                      $y_{m+1} = x_{m+1}$ if σ[ $O_i$, otherwise $y_{m+1} = x'_{m+1}$}
          Conf := {$t_1$, $t_2$, $t_3$, ....$t_{m+1}$}          {Conf is determined through the use of the output
                                                                sequence R and the corresponding input subsequence of $tc_i$}
                                                                {$t_{m+1}$ is a symptom transition}
          $Plus_i$ := $Plus_i$ - σ
          Forall σ" [ $Plus_i$ do
            If (Common (σ", $σ_f$) = Common (σ, $σ_f$)) then
            $Plus_i$ := $Plus_i$ - σ"
          Endforall
        Endforall
        If ($Conf_i$ ( Conf □ □) then
                $Conf_i$ := $Conf_i$ ( Conf
        Else $Conf_i$ := Conf
    Until ($Plus_i$ = □)
```

Each execution of ComputeConf generates the minimal conflict set for the corresponding considered symptom.

Step 5: Generation of diagnostic candidates and their diagnoses

Diagnostic candidates are transitions which are suspected to be faulty. Therefore, each one of them should have a non empty intersection with each conflict set. It also has to be consistent with all observations.

Step 5A: Generation of the initial tentative candidate set

Algorithm: The initial tentative candidate set **"ITC"** will be formed by the **intersection** of all conflict sets. Each element T_k in ITC represents a tentative candidate transition (with an output or a transfer fault) which may explain all symptoms.

Step 5B: The FTC, the ending state and the ustset sets

Algorithm: For the generated initial tentative candidate set ITC, if there is a unique symptom transition **"ust"**, it will be contained in the ITC. In that case, we split the ITC into the set **"ustset"** which will initially contain the ust and the final tentative candidates set for transitions with transfer faults **"FTCtr"** which will contain the rest of transitions in ITC. Otherwise, the full ITC set forms the FTCtr set and the ustset is kept empty.

If the ust is contained in the ustset, it will be processed as follows. All test cases in the initially given test suite "TS" are scanned for transitions that are equal to the ust. If for all found transitions their corresponding observed outputs is equal to the uso and for the remaining transitions in the corresponding test cases all observed and expected outputs are equal, which means the ust explains all observations, then the ust is considered a diagnostic candidate for an output fault.

```
Procedure ust-processing (ustset)
Forall $tc_m$ □ TS  DO
    Forall $i_{m,n}$ □ $tc_m$ DO                      {if for $tc_m$ corresponds r paths, then $T^k_{m,n}$ is the
        IF  ($T^k_{m,n}$ = ust) THEN              transition which corresponds to the n-th input in $tc_m$
            IF ($ô^k_{m,n}$ <> uso) THEN    and the n-th output in the k-th sequence of $\hat{O}_m$}
                ustset = □; exit                {the ust is not a diagnostic candidate}
            ELSE IF $o^k_{m,n+l}$ <> $ô^k_{m,n+l}$ THEN    {l =1, 2, ..., $i_m$ where n +$i_m$ is the
                                                           length of the test case $tc_m$}
                                              ustset = □; exit
    ENDForall
ENDForall
```

For each transition T_k in the FTCtr, we compute the set of all faulty transfer states called **"EndStates$_k$"**, to which T_k might transfer. For each transition, we consider all states in the machine, with the exception of the expected NextState of T_k, one at a time. For each state s under consideration, s will be included in EndStates$_k$, if under the assumption that s is the

NextState of T_k, the expected and observed outputs are equal for all succeeding transitions in all test cases.

```
Procedure findendingstates (FTCtr);
    Forall $T_k$ in FTCtr Do                         {$T_k$ is the k-th transition in FTCtr }
    EndStates$_k$ := □           {EndStates$_k$ is the set of all states to which $T_k$ might transfer }
        Forall state s □ S and s □ NextState($T_k$) Do   {NextState is a function which offers the next state
                flag := true                        for any transition in the specification machine}
                Forall tc$_m$ □ TS  Do               {if for tc$_m$ corresponds r paths, then $T^k_{m,n}$ is the
                    Forall i$_{m,n}$ □ tc$_m$ Do    transition which corresponds to the n-th input in tc$_m$
                    IF  ($T^k_{m,n}$ = $T_k$ ) THEN      and the n-th output in the k-th sequence of $\hat{O}_m$}
                      NextState'($T^k_{m,n}$) = s;      {let the ending state of  in the specification be s}
                      Apply the test case tc$_m$ to the modified specification
                      IF (newly set of expected output sequences <> set of observed outputs)
                      THEN  flag := false; exit
                    ENDForall
                ENDForall
                IF (flag = true) THEN
                    EndStates$_k$ := EndStates$_k$ ≈ {s}
        ENDForall
    ENDForall
```

Step 5C: Identification of diagnostic candidates and generation of diagnoses

Algorithm: In this step we remove all correct (i.e. transitions with empty ending state sets or empty outputs sets) transitions from the final tentative candidate set. All transitions in the resulting **"DCtr"** set (if not empty) are diagnostic candidates with transfer faults. For each transition T_k in the DCtr and for each state s_{ik} in the EndStates$_k$, a diagnose, stating that T_k might transfer to state s_{ik}, is generated. An extra diagnose, stating that the ust might have an output fault, is also generated, if the ustset is not empty.

Step 6: Additional diagnostic tests

Depending on the results of the previous steps, the following different possibilities might be present.

Case 1: The ustset contains the ust transition and the DCtr is empty. In such a case, the ust is the faulty transition with the output fault uso and no further diagnostic tests are required.

Case 2: The ustset is empty and the DCtr is a singleton with a corresponding singleton ending state set. In such a case, the only transition of DCtr has a transfer fault to the state in the corresponding ending state set. No further tests are required.

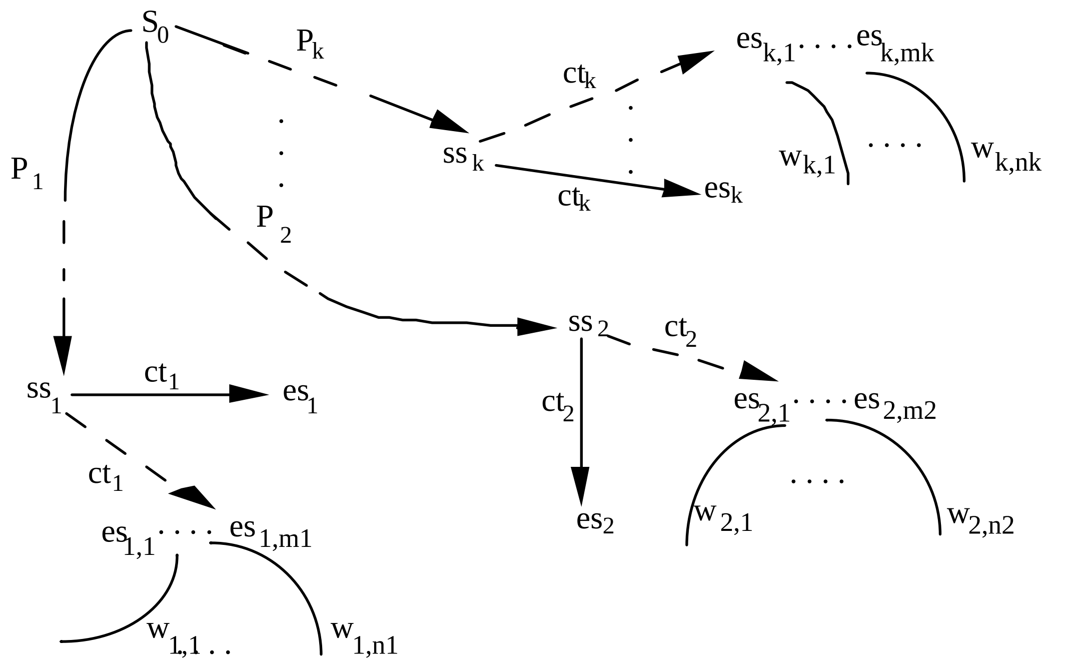
Case 3: The ustset is empty and the DCtr is a singleton with a corresponding ending state set with more than one element, or the DCtr has more than one element. Therefore, each element in DCtr might be the faulty transition with a transfer fault. In such a case, we should process the elements of DCtr to derive further tests with the purpose of identifying the faulty transition and the state to which it transfers.

Algorithm for Case 3:
For each transition T_k in DCtr, additional test cases have to be selected and executed, in order to be able to know exactly to which state it transfers. These test cases should have the ability of distinguishing between the different states contained in the corresponding ending state set "EndStates$_k$" and possibly the correct ending state of the transition. Therefore, a **"limited characterization set"** W$_k$ has to be computed for the states in EndStates$_k$ and the correct state. A limited characterization set is different from the characterization set defined in [Chow 78], since it concerns only a subset of the states rather than the whole set of states in the machine. It is formed by sequences of inputs such that if they are applied to the machine in one of the states in EndStates$_k$, some produced outputs will be different from the outputs obtained if the same input sequences were applied to the machine in any other state of EndStates$_k$ or the correct state. Each of these additional test cases is a concatenation of an input sequence, called transfer sequence, required to take the machine from its initial state to the starting state of T_k, the input for T_k and a sequence of inputs from the W$_k$.

In order to avoid any ambiguities, the transfer sequence and the limited characterization set should be chosen in such a manner that they do not involve any candidate transition in the DCtr set. Figure 2 illustrates the progressive construction of the additional test cases needed to distinguish the faulty transition from the rest of the diagnostic candidates of DCtr.

The construction of the additional tests is progressive because if the fault is located, the rest of the additional tests need not be generated, since we assume the single fault hypothesis. If some of the generated tests are already included in the initially given test suite, this will be taken into consideration for the analysis of the obtained outputs, but they need not be applied again to the IUT. If the application of these additional tests generates the expected outputs, the transition is declared correct and is removed from the corresponding diagnostic candidates set. When a faulty transition is found, the analysis of observed outputs identifies the faulty transfer of that transition and the search is stopped.

Case 4: The ustset contains the ust transition and DCtr is not empty. In such a case, we first check the ust transition by generating for it an additional test case. Such a test case should be selected with the restriction that it does not imply the execution of any transition in DCtr. If its application generates the expected set of output sequences, then the ust is declared correct and the search for the faulty transition in DCtr has to be done as in Case 3. Otherwise, ust is the transition with an output fault and the search is stopped.



Notes:
1) s_i's, ss_i's, es_i's and $es_{i,j}$'s are states in the machine.
2) ct_i's are candidate transitions
3) each P_i is a transfer sequence from the initial state to the starting state of ct_i
4) The input sequence in each path in the tree represents a possible additional diagnostic test for a specific candidate transition and a specific ending state.
5) Solid lines transitions indicate the predicted behavior of the diagnostic candidates (no fault case).
6) Each subtree starting with P_i represents additional tests for transition tc_i
7) The set of sequences $\{ w_{i,1} \ldots w_{i,ni} \}$ distinguishes between the states $es_{i,1} \ldots es_{i,mi}$ in the $EndStates_i$ for candidate transition ct_i

Figure 2 : Construction of additional diagnostic tests

The above cases are covered by the following algorithm:

```
IF (ustset = {ust} AND DCtr = □)
THEN
   Print "The ust transition has an output fault"
ELSE    IF ustset = □ AND DCtr = {$T_1$ } AND EndStates$_1$ = {$s_1$}
        THEN
           Print "$T_1$ is the faulty transition with transfer fault to $s_1$"
        ELSE    IF ustset = $\emptyset$ AND DCtr = {$T_1$ , ..., $T_d$ }
                THEN
                   Findtransferfault (DCtr)
                ELSE    IF ustset = {ust} AND DCtr = {$T_1$ , ..., $T_d$ }
                        THEN
                                select test cases for the ust;
                                apply these tests to the IUT;
                                IF (observed set of output sequences<> expected set of
                                     output sequences)
                                THEN
                                Print "The ust has an output fault and all other
                                            transitions are correct"
                                ELSE Findtransferfault (DCtr)
```

```
Procedure Findtransferfault (DCtr)
        flag := false; k := 1;
        REPEAT          {$T_k$ is a transition in DCtr}
                select diagnostic tests for $T_k$ ;
                apply these tests to the implementation;
                IF (observed set of output sequences <> expected set of output sequences)
                THEN
                   flag := true;
                  Print "$T_k$ has a transfer fault. Its ending state is deduced from the analysis of the
                    observed outputs. All other transitions are correct"
                  ELSE Print "$T_k$ is correct"
                k := k + 1
        UNTIL (flag = true)
```

4. An application example

Suppose that the following initial test suite for the ONFSM specification shown in Figure 1, is given:
TS = {rabb, rabaa, rbabab}

Steps 1 and 2: The application of this TS to the specification of Figure 1 and the implementation of Figure 3, yields the expected and observed output sequences, as shown in Table 1.

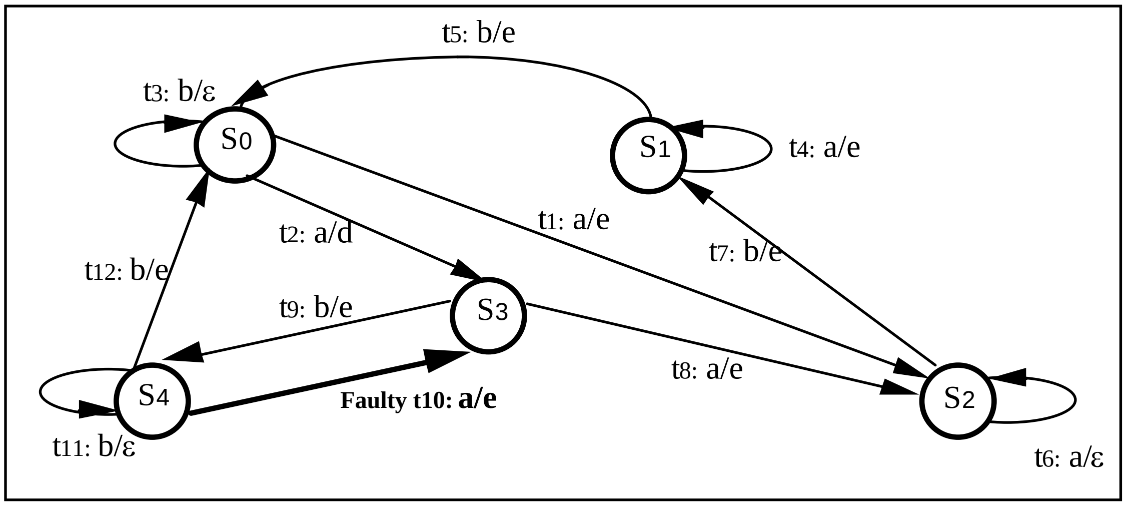


Figure 3: A faulty implementation I

tc1	r a b b
Corresp. spec. seqs of transitions	{tr t1 t7 t5 , tr t2 t9 t11 , tr t2 t9 t12}
Expected sequences of outputs	{ε e e e , ε d e ε , ε d e e}
Observed sequences of outputs	{ε e e e , ε d e ε , ε d e e}

tc2	r a b a a
Corresp. spec. seqs. of transitions	{tr t1 t7 t4 t4 , **tr t2 t9 t10 t6**}
Expected sequences of outputs	{ε e e e e , **ε d e e ε**}
Observed sequences of outputs	{ε e e e e , **ε d e e e**}

tc3	r b a b a b
Corresp. spec. seqs of transitions	{tr t3 t1 t7 t4 t5 , tr t3 t2 t9 t10 t7}
Expected sequences of outputs	{ε ε e e e e , ε ε d e e e}
Observed sequences of outputs	{ε ε e e e e , ε ε d e e e}

Table 1: Test cases and their outputs

In Table 1, a reset transition tr is assumed to be available for both the specification and the implementation. We use the symbol "r" to denote the input for such a transition and the symbol "ε" to denote its output.

Step 3: A difference between observed and expected outputs is detected for test cases tc_2. Therefore, the symptom is:

$Symp_2 = (O_2 \neq \hat{O}_2)$

Step 4: Corresponding to the above symptom, the application of the algorithm in Step 3 will progress as follows:

$Plus_2 = O_2 - \hat{O}_2 = \{\varepsilon dee\varepsilon\}$ $Checkset_2 = \hat{O}_2 - O_2 = \{\varepsilon deee\}$

The comparison of the above two sets implies the following conflict set:

$Conf_2 = \{t_2, t_9, t_{10}, t_6\}$

$Conf_2$ consists of all those specification transitions which can be identified by the input sequence tc_2 and the specification output sequence in $Plus_2$. Since there is only one conflict set, t_6 is at the same time a symptom transition and the ust transition.

Step 5A: Since there is only one conflict set, no intersection is needed. The initial set of tentative candidates is the following:

$ITC = \{t_2, t_9, t_{10}, t_6\}$

Step 5B: Generate the corresponding FTCtr and the ustset sets:

$FTCtr = \{t_2, t_9, t_{10}\}$, $ustset = \{t_6\}$

The processing of the above sets and the computation of the ending state sets for the transitions in FTCtr leads to:

$ustset = \{t_6\}$,

$EndStates[t_2] = \{\}$, $EndStates[t_9] = \{\}$, $EndStates[t_{10}] = \{s_1, s_3\}$

Step 5C: The transitions with empty ending state sets are correct, therefore they are removed from the final tentative candidate set. The resulting diagnostic candidates sets are the following:

$DCtr = \{t_{10}\}$, $ustset = \{t_6\}$, With the use of the ending state sets and the ustset generated in Step 5B, the following diagnoses are deducted:

Diag1: t_6 might have an output fault of e instead of ε.

Diag2: t_{10} might transfer to state s_0 instead of state s_2.
Diag3: t_{10} might transfer to state s_3 instead of state s_2.

Step 6: In order to reduce the space of the resulted diagnoses, additional diagnostic tests have to be selected. Since output faults are in general easier to be tested and require less tests, we start with Diag1. A possible test for t_6 is "tc_{a1} = raa". Since the corresponding expected and observed sets of outputs are equal (i.e., $O_{a1} = \hat{O}_{a1}$ = {εde, εeε}), t_6 is confirmed to be correct and t_{10} is confirmed to have a transfer fault. But in order to know to which state t_{10} transfers, another test is needed. A possible test for distinguishing between the two states s_1 and s_3 is "tc_{a2} = rabaaa". The application of such a test case generates the following set of output sequences: {εeeeee, εdeeeε}. Such a result confirms that t_{10} transfers to state s_3 and not to state s_1.

5. Concluding discussion

In this paper, we extended the applicability of the diagnostic approach proposed in [Ghed 92] to a new class of non-deterministic systems. We assume that the software system specifications and implementations can be represented by observably non-deterministic finite state machines. For a ONFSM, a sequence of input and one of its corresponding output sequences identify exactly a unique path (a specific sequence of transitions) in the machine. Therefore, it will be relatively easy to localize the fault, since we need to deal only with that specific group of transitions where the fault has been detected. On the other hand and if the general non-deterministic FSM model is used, multiple paths can correspond to a single pair of input and output sequences. such a multiplicity of paths makes the fault localization problem even harder, since for a single observed symptom, the search for the fault will be spread to a set of paths rather than a single one.

Most steps of the proposed algorithm were modified (with respect to [Ghed 92]) to accommodate the newly defined ONFSM model. In such a model, different paths might be executed by a single input test sequence. The occurrence of a fault in one of the implementation transitions might modify the observed set of output sequences, which corresponds to a given test case. As a result, new output sequences might be observed, while other expected sequences might be missed. To deal with such a problem, we introduced a new algorithm for the conflict set generation, where we consider the different possible inclusion relations the sets O_i and $\hat{O}_i$ might have. Similar changes were also included in the remaining steps of the global diagnostic algorithm.

Many important questions are left for future work, such as the diagnostic of systems which are represented by general non-deterministic FSMs. Another important question, is the diagnostic of systems, represented by different models (i.e., FSMs, communicating FSMs, extended FSMs,...), and which allow multiple faults. Such a question is known to be a very difficult one. A possible starting point is to try to solve it for at least some special classes of multiple faults such as the presence of two (output plus transfer) faults in the same transition of the implementation machine.

Acknowledgments: The authors would like to thank F. Khendek for useful discussions on the NFSMs model used in the paper. They also would like to thank R. Gotzhein and R. Groz for their useful comments**.** This work was partly supported by the Natural Sciences and Engineering Research Council of Canada, the Ministry of Education of Québec and the IDACOM-NSERC-CWARC Industrial Research Chair on Communication Protocols.

References

[Boch 91] G.v. Bochmann, R. Dssouli, A. Das, M. Dubuc, A. Ghedamsi, and G. Luo, "Fault models in testing", Invited paper in 4-th IWPTS, Leidschendam, Holland, 15 - 17 Oct. 1991.

[Cern 92] E. Cerny, Verification of I/O trace set inclusion for a class of non-deterministic finite state machines, submitted to CAV'92, Montréal, Canada.

[[Chow 78] T.S. Chow, "Testing Design Modelled by Finite-State Machines", IEEE Trans. S.E. 4, 3, 1978.

[Davi 88] R. Davis, and W. Hamscher, "Model-based reasoning: Troubleshooting", in: Exploring Artificial Intelligence, edited by Shrobe, H. E. and the American Association for Artificial Intelligence, pp. 297-346, Morgan Kaufman, 1988.

[Fuji 91] S. Fujiwara, G.v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test selection based on finite state models", IEEE Trans. on Software Engineering, Vol. 17, No. 6, June 1991, pp. 591-603.

[Gene 84] M.R. Genesereth, "The use of design descriptions in automated diagnosis", Artificial Intelligence 24(3), 1984, pp. 411-436.

[Ghed 92] A. Ghedamsi and G.v. Bochmann, "Test result analysis and diagnostics for finite state machines", accepted at the 12-th international conference on distributed systems, Yokohama, Japan, June 9-12, 1992.

[Gone 70] G. Goenenc, "A method for the design of fault detection experiments", IEEE Trans. Computer, Vol. C-19, pp. 551-558, June 1970.

[Klee 87] J. de Kleer, and B.C. Williams, "Diagnosing multiple faults", Artificial Intelligence 32(1), 1987, pp. 97-130.

[Koka 90] K.C. Ko, "Protocol test sequence generation and analysis using AI techniques", Master thesis, Dept of Comp. Sci., UBC, Jul. 1990.

[Kore 88] B. Korel, "PELAS-Program error-locating assistant system", IEEE Trans. on Software Engineering, Vol. 14, No. 9, September 1988.

[Luog 92] G. Luo, G.v. Bochmann, M. Yao and A. Ghedamsi, "Test Generation based on Nondetermistic Finite State Machine", in preparation.

[More 90] L. J. Morell, "A theory of fault based testing", IEEE Trans. on Software Engineering, Vol. 16, No. 8, August 1990.

[Nait 81] S. Naito and M. Tsunoyama, "Fault Detection for Sequential Machines by Transition-Tours", Proc. of FTCS (Fault Tolerant Computing Systems), pp.238-243, 1981.

[Reit 87] R. Reiter, "A theory of diagnosis from first principles", Artificial Intelligence 32(1), 1987, pp. 57-96.

[Sabn 88] K.K. Sabnani and A.T. Dahbura, "A protocol Testing Procedure", Computer Networks and ISDN Systems, Vol. 15, No. 4, pp. 285-297, 1988.

[Scho 76] E.H. Shortlife, "Computer-based Medical Consultations : MYCIN", Elsevier, New-York, 1976.

[Stru 89] P. Struss, and O. Dressler, "Physical Negation - Integrating Fault Models into the General Diagnostic Engine", Proceedings IJCAI, Detroit - Michigan, 1989, pp. 1318-1323.

[Ural 87] H. Ural, "A Test Derivation Method for Protocol Conformance Testing", Proc. of the 7th IFIP Symposium on Protocol Specification, Testing and Verification, Zurich, May 5-8 1987.

[Vuon 90] S.T. Vuong and K.C. Ko, "A novel approach to protocol test sequence generation", IEEE Global telecomm. conference and exhibition, San Diego, California, Dec. 2-5, 1990, vol. 3, 904.1.1 - 904.1.5.