REVERSE-ENGINEERING OF COMMUNICATION PROTOCOLS

David Lee (lee@research.att.com)
Krishan Sabnani (kks@research.att.com)

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, New Jersey 07974

ABSTRACT

We study the problem of locating the differences between a protocol specification and its implementation. A solution to this problem will be useful in reverse-engineering of proprietary protocol standards. We give an exact procedure for solving this problem. If there is only one difference between the implementation and the specification then the algorithm will locate the difference and, therefore, identify the implementation machine. Otherwise, it will detect that the implementation machine has more than one change. The run time of our algorithm is a low-degree polynomial in the number of states and inputs of the machine. In the published literature only heuristic procedures have been reported. We first describe briefly a brute-force version of the algorithm with a cost $O(pn^5)$ *where* n *is the number of states of the specification machine and* p *is the number of inputs. We then present a fast algorithm with a cost* $O(pn^3 \log n)$*. We also give an improvement for which the cost on the average is* $O(pn^2 \log n)$*. Finally, we describe a heuristic procedure that uses a test of comparable length to a conformance test sequence which has been used successfully in practice.*

1. INTRODUCTION

A lot of research work has been done on testing of communication protocols [DSU90, UP83, SD88, ML90, ADLU91, MP91, WL92]. We check if a protocol implementation conforms to the specification, or, equivalently, if both of them have the same input/output (I/O) behavior. To check conformity, a test sequence which exercises different parts of the protocol behavior is used. We have developed some automated test generation techniques which have been used extensively in practice. These procedures typically result in a 3-to-1 reduction in the test sequence length compared to ad hoc procedures used in the past.

The purpose of conformance testing is only to find out if an implementation is different than its specification. An interesting yet more complex problem is how to locate the differences between a protocol specification and its implementation if they are found to be different. We would call this problem *reverse-engineering*. A solution to this problem has various applications. For example, it can make easy the job of correcting a protocol implementation so that it conforms to its specification.

Sometimes, it is essential to keep track of proprietary protocol standards by observing the behavior of their implementations. This is especially important for designers of protocol implementations which have to interoperate with proprietary protocol implementations. For example, a segment of computer industry manufactures channel extenders for mainframes. A channel extender enables a remote peripheral to communicate with a mainframe. To keep designs of these extenders up-to-date, designers have to keep track of the protocols used in the mainframes. Manufacturers of mainframes are usually slow or reluctant to inform its users about the changes in these protocols. We need a procedure which can enable us to locate changes in these protocols by observing the I/O behavior of these mainframes.

Here is an informal description of this reverse-engineering problem: given a formal specification of a protocol entity as one finite state machine A, we want to find out the underlying finite state machine model of a protocol implementation B. We assume that the specification and implementation machines have only one difference in their state transitions (next state or output). Such a difference will be called a *change* or *fault* [(1)]. For our application, the single-change assumption is not too restrictive; we could monitor the implementation machine frequently before multiple changes occur.

If the number of changes is larger than one then we have the general reverse-engineering problem, which is similar to the machine identification problem [Mo56]. It is known to have exponential complexity and has few practical applications. As we shall show later in [LS93], if there are only a constant number of changes, it still takes polynomial time to identify the implementation machine. On the other hand, learning with Valiant's PAC model is a well-studied topic in theoretical computer science [Va84, An92]. In that model, a learner often gets hints from a teacher and finally identifies the object to be learned. For our application, there is only one difference between A and B and we want to take advantage of their structural similarities and identify (reverse-engineer) B using a test of practical length. This problem has been studied in [GB92, GBD93], and some heuristic procedures were reported there.

In this paper, we present a method such that with some

(1) For the application of reverse-engineering, it is not a "fault" but a "change", which we want to locate and identify. In the rest of the paper, we use "change".

0-8186-3670-X/93 $03.00 © 1993 IEEE

usual assumptions as in conformance testing it is guaranteed that we can locate the difference if there is only one change. We can also conclude for sure the presence of multiple changes. Our exact procedure has two parts: (1) Determine if B is different than A; (2) If the first part is true then locate the difference and identify B. Part (1) is essentially a conformance testing (also called fault-detection) problem and has been well-studied. Part (2) is more complex; we guess all the possible machines for B and verify our conjecture. Instead of a brute-force enumeration, we develop efficient algorithms. We have an $O(pn^3 \log n)$ algorithm, where n is the number of states and p is the number of inputs. We also give an improvement with an average cost $O(pn^2 \log n)$.

The exact procedures can generate test sequences which are too long to be useful in practice; it is necessary to develop some heuristic procedures which generate significantly shorter test sequence. We present such a heuristic procedure here, which is a modification of the one in [SD88]. After applying a test sequence to the implementation, we analyze the test results to locate the change.

In Section 2, we present the FSM model used here. In Section 3, we present an exact algorithm for locating single-changes in protocol implementations. In Section 4, we present a heuristic procedure. Finally, in Section 5, we discuss multiple changes and conclude the paper.

2. FINITE STATE MACHINE MODEL

In the section, we give the notation for finite state machines used in this paper.

Definition 2.1. A finite state machine (FSM) M is a quintuple

$$M = (I, O, S, \delta, \lambda)$$

where I, O, and S are finite and non-empty sets of input symbols, output symbols, and states, respectively;

δ: $S \times I \rightarrow S$ is the state transition function;

λ: $S \times I \rightarrow O$ is the output function.

□

Such machines are usually called *Mealy* machines in the literature [Ko78]. There is a variant, the *Moore* machine, in which the output is determined only by the state. These machines form a subclass of Mealy machines. All results for the Mealy machines apply to Moore machines as well.

Obviously, FSM's in our model are deterministic. We assume that an FSM has a unique initial state, denoted as s_1.

Let M be a finite state machine. We extend the output function λ and the transition function δ of the machine from input symbols to strings as usual: For an initial state s_1 and an input sequence $x = a_1,...,a_k$, denote the corresponding output sequence by $\lambda(s_1,x)$ and the final state by $\delta(s_1,x)$; that is, $\lambda(s_1,x) = b_1,...,b_k$, where $b_i = \lambda(s_i,a_i)$ and $s_{i+1} = \delta(s_i,a_i)$ for $i = 1,...,k$, and the final state is $\delta(s_1,x) = s_{k+1}$.

A finite state machine is *minimized* (or reduced) if no pair of its states are equivalent; that is, for every pair of states s_i, s_j, $i \neq j$, there is an input sequence x that distinguishes them: $\lambda(s_i, x) \neq \lambda(s_j, x)$. A classical state-partitioning algorithm [Mo56, Ho71] determines state equivalence and decides whether the FSM is minimal.

We can represent an FSM by a directed graph whose nodes are the states and whose arcs are transitions labeled by the associated I/O pairs. We say that an FSM is *strongly connected* if its graph is; that is, for every pair of states (s_i, s_j), there is an input sequence x such that $\delta(s_i,x) = s_j$.

We make the following assumptions.

[1] The specification FSM A is strongly connected. The reason for this assumption is that, if A is not strongly connected, and if in the experiment the implementation machine B starts in a state from which some other states can not be reached, then we will not be able to visit all states of B during the test. Thus, we will not be able to tell with certainty about B.

[2] The specification machine A is minimal.

[3] The implementation machine B does not change during the experiment and has the same input and output alphabet as A.

We say that machine A has a *reset capability* if there is an input symbol r that takes the machine from any state of A back to the initial state, i.e., $\delta_A(s_i,r) = s_1$ for all states s_i. We say that the reset is *reliable* if it is guaranteed to work properly for the implementation machine B, i.e., $\delta_B(s_i,r) = s_1$ for all s_i. We further assume:

[4] The implementation machine B has a reliable reset r, a special input symbol; upon input r, the machine B moves back to its initial state s_1.

In addition, an upper bound must be placed on the number of states of B. Otherwise, no matter how long our test is, it is possible that the test will not exercise the "changed" part of B. The usual assumption made in the literature, and which we will also adopt, is that the changes do not increase the number of states of the machine, that is:

[5] The implementation machine B has no more states than A.

3. AN EXACT REVERSE-ENGINEERING PROCEDURE

Given a specification machine A and its implementation B, we want to know if there are any changes (differences) in B by observing its I/O behavior. There are two types of changes: "output changes", i.e., one or more transition may produce outputs different than A, and "next-state changes", i.e., transitions may go to states different from A.

We now present an exact procedure, which identifies the implementation machine if it has only a single change. It has two steps:

[1] Check if there are any changes, or, equivalently, if B conforms to A;

[2] If B is different than A then locate the changes and identify B.

Step [1] is a conformance testing, and there are a number of methods available. Our emphasis is on Step [2], locating the changes.

For Step [1], we use a checking sequence of length $O(pn^3)$. For Step [2], we first give a brute-force algorithm

which generates a test sequence of length $O(pn^5)$. Then, we give a fast algorithm which generates a test sequence of length $O(pn^3 \log n)$. Finally, we give an improvement by a factor n on the average.

3.1. DETERMINING THE PRESENCE OF CHANGES

For Step [1] of our exact procedure, a *checking sequence* (for conformance testing or fault-detection) is sufficient. Here is a definition of a checking sequence.

Definition 3.1. Let X be a specification FSM with n states and initial state s_1. A *checking sequence* for X is an input sequence x that distinguishes X from all other machines with no more than n states; i.e., every machine Y with at most n states that is not isomorphic to X produces on input x a different output than that produced by X starting from s_1.

□

All proposed methods for fault-detection have the same basic structure. We want to make sure that every transition of the specification FSM A is correct in the implementation FSM B. For every transition of A, say from state s_i to state s_j on input a, we want to apply an input sequence that transfers the machine to s_i, apply input a, and then verify that the next state is s_j. The methods differ by the types of subsequences they use to verify that the machine is in the right ending state. Hennie proposed a method that uses (preset) distinguishing sequences [He64], and it was observed by Kohavi's [Ko78, KK68] that the same method essentially would work also with an adaptive distinguishing sequence. Sabnani and Dahbura proposed a method using UIO sequences, see [DSU90, SD88, ADLU91] and other papers in the references. Polynomial-time algorithms have been obtained recently [LY93,YL93].

Hennie proposed a second (but in general exponential) method in [He64] that uses a type of sequence that always exists. Such a test sequence, a *characterization set* of an FSM M, is a set C of input sequences such that every pair of states is distinguished by at least one member of C. For every pair s_i, s_j of states there is a sequence x in C such that $\lambda(s_i,x) \neq \lambda(s_j,x)$. Every reduced FSM has a characterization set consisting of no more than $n-1$ sequences, where each sequence has length no more than $n-1$. Such a set can be constructed by the classical state-minimization algorithm. Characterization sets were used in [Vas73, Ch78] to obtain polynomial-time algorithms of checking sequences for machines with reset. We use their algorithm for Step [1] of our exact procedure, and we describe it briefly here.

The main idea of the Vasilevskii-Chow algorithm is as follows. We first identify the n states of the implementation B by separating them from one another. We then verify all the transitions, their outputs, and next states. The algorithm depends on the reliable reset capability of B; at any moment we can take the machine back to the initial state s_1 by the input r.

Starting with the initial state s_1 of A, we construct a spanning tree rooted at s_1 (depth-first search tree, for example). We identify the nodes (states) and tree edges (transitions) in the order they are traversed. This procedure is carried out inductively. Suppose that we have identified i states, $s_1, \cdots, s_i$, and the $i-1$ tree edges leading to the i states. We now want to identify the ith tree edge e_i leading to the $(i+1)$st state s_{i+1}. We first apply the input associated with the edge e_i and verify its corresponding output. If there is a discrepancy of the outputs from A and B then a changed (output) is detected. Otherwise, we verify the ending state s_{i+1} of e_i as follows. For each identified state s_j, $1 \le j \le i$, we find a sequence in the characterization set C that distinguishes s_j from s_{i+1}. We take the machine to state s_j along the identified tree edges and apply this sequence; we then take the machine to state s_{i+1} along the identified tree edges and apply the same input sequence. If we observe expected outputs for all the identified states s_j then we know that we have identified the $(i+1)$st state along with the tree edge leading to that state. After we have identified all n states and the tree edges without observing any discrepancies, we proceed to identify non-tree transitions in a similar manner. For the details, see [Vas73, Ch78]. As a result, we obtain a checking sequence, which consists of pn^2 input sequences of length no more than $2n-1$, interposed with reset:

$$r\, x_1\, r\, x_2\, r \cdots r\, x_{pn^2} . \tag{3.1}$$

The total cost is $O(pn^3)$. We call this set of pn^2 test sequences the *basic test set.*

For our purposes, a change is detected if a discrepancy is observed from applying a sequence x_i in the basic test set; we know that the sequence x_i (of length no more than $2n$) has caught the change, and we only have to examine only these $2n$ transitions to locate the change. This will be discussed next.

3.2. LOCATING THE CHANGE

The basic test set provides a checking sequence in (3.1), and a next-state or output change can be detected by at least one of its pn^2 input sequences. We denote such a sequence as

$$x = I_1 , \cdots , I_{2n} . \tag{3.2}$$

The corresponding output sequences from A and B are

$$\begin{aligned} \lambda_A(s_1, x) &= O_1, \cdots, O_{2n}, \\ \lambda_B(s_1, x) &= \tilde{O}_1, \cdots, \tilde{O}_{2n}. \end{aligned} \tag{3.3}$$

We assume that the earliest discrepancy of the two output sequences occurs at the kth output symbol, where $1 \le k \le 2n$, and we have

$$\begin{aligned} \lambda_A(s_1, x) &= O_1, \cdots, O_{k-1}, O_k, \\ \lambda_B(s_1, x) &= \tilde{O}_1, \cdots, \tilde{O}_{k-1}, \tilde{O}_k \end{aligned} \tag{3.4}$$

where $O_j = \tilde{O}_j, j = 1, \cdots, k-1$, and $O_k \neq \tilde{O}_k$.

Since B has only a single change, it must be exercised by the input sequence x during one of the first k transitions, *not necessarily the kth one.* We denote the prefix of x of length k by x_k. Without loss of generality, we assume that x_k carries machine A from state s_1 to $s_2, s_3, \cdots, s_{k+1}$, where s_1 is the initial state and these $k+1$ states may not be different.

There are three types of single changes: (1) An input symbol carries the implementation machine B to the correct state but produces a different output than from A; (2) An input

symbol carries the implementation machine B to a different state than in A but produces a same output; (3) An input symbol carries the implementation machine B to a different state than in A and also produces a different output. Note that both the next-state and output changes in (3) are associated with a single transition in B. Note that from (3.4), Type (2) changes can only occur during the first $k - 1$ transitions and Type (3) the last (kth) transition.

Our algorithm runs in two steps. First, we assume that the machine has a Type (1) output change, and verify it. If this is not the case, we assume that the machine has a Type (2) or (3) next-state change in the first k transitions, and verify it. If that is not the case either, we conclude that machine B has multiple changes.

3.2.1. OUTPUT CHANGE

We first assume that B has an output change. Since B only has a single change, which only produces one different output; as defined earlier, it must be the kth output. We modify the specification A: we change the output symbol to $\tilde{O}_k$ for the transition from s_k to s_{k+1} upon input I_k. The modified specification is denoted by $\tilde{A}$. All the assumptions on A remain the same for $\tilde{A}$. We conduct a checking experiment on B with respect to $\tilde{A}$. If B conforms to $\tilde{A}$, then we have verified that the implementation B is isomorphic to $\tilde{A}$. Therefore, we have reverse-engineered B, which is identified to be $\tilde{A}$. The run time is that of a checking experiment: $O(pn^3)$.

If B does not conform to $\tilde{A}$, then it has a single next-state change or multiple changes. We proceed to the following step.

3.2.2. NEXT-STATE CHANGE

If there is a single next-state change, then it can occur in any one of the first k transitions from state s_j to s_{j+1}, $j = 1, \cdots, k$. We assume that the change occurs at the jth transition, $j = 1, \cdots, k$, and verify each assumption in turn. A successful verification of the assumption locates the change and identifies the implementation machine. Otherwise, we conclude that there are multiple changes.

Upon input I_j, the jth transition is supposed to be from s_j to s_{j+1}, generating output O_j. However, B goes from s_j to $\tilde{s}_{j+1}$, producing $\tilde{O}_j$. State $\tilde{s}_{j+1}$ can be any of the $n - 1$ states except the right one, and we verify each of them in turn. More specifically, suppose that we conjecture that a next-state change occurs at the jth transition. We assume that input I_j takes B to state s_r instead s_{j+1} where s_r can be any of the $n - 1$ states except the right state s_{j+1}. For each of the $n - 1$ possibilities, we modify the specification A; upon input I_j, the modified machine moves from state s_j to s_r and outputs $\tilde{O}_j$. We then verify that B conforms to the modified specification. This can be done in time $O(pn^3)$ using a checking sequence.

There can $n - 1$ different next states as $\tilde{s}_{j+1}$. Therefore, there are no more than $2n^2$ possible machines as B. We run a checking experiment for each possible machine with a cost $2pn^3$. Therefore, the total run time of the algorithm is $O(pn^5)$. This is too costly to be practical. We present next a fast algorithm with a cost $O(pn^3 \log n)$.

3.2.3. A FAST ALGORITHM

The brute-force algorithm is expensive; it examines $2n^2$ conjectured machines and runs a checking experiment for each one, resulting in a cost $O(pn^5)$. The overhead for each machine is $O(pn^3)$. Instead, we can cross-verify these $2n^2$ machines with an overhead $O(pn \log n)$ for each machine, resulting in an $O(pn^3 \log n)$ algorithm. We use cross-verification to rule out wrong conjectures of the implementation machine more efficiently than verifying each one of them separately. More specifically, we take two conjectured machines A_1 and A_2. If they are isomorphic then we discard one of them. Otherwise, we find an input sequence such that the two machines will produce different output sequences. We also apply this input sequence to the implementation machine B. Then the output sequence from B should be different than at least one of the outputs from A_1 and A_2, and we can easily rule out the one with a different output; it cannot be isomorphic to B. We repeat the procedure to rule out all the conjectured machines which are not isomorphic to B. The following is a detailed description of our approach.

As before, we find the input sequence x from the basic test set (3.2) that reveals the changes of machine B as an output discrepancy. We first verify whether it is an output change. If that is the case, then we can easily identify the change and we are done. Otherwise, there is a next-state change. We examine the k transitions from input sequence $x_k = I_1, \cdots, I_k$, which takes the machine from state s_1 to $s_2, s_3, \cdots, s_{k+1}$, successively. Note that one and only one of them can go to a different state from the original specification A; but that may not be in the kth transition. For the jth transition with input I_j from state s_j to s_{j+1}, $j = 1, \cdots, k$, we conjecture that it has been modified. We then either verify or rule out our conjecture. If we can verify one conjectured change, then we have fulfilled our goal. If we rule out all the conjectures after considering all the k transitions, then machine B must have more than one change. We process each of the k transitions in the following three steps: (1) Enumerate all possible changed next states and the corresponding changed machines; there are $n - 1$ of them in total. (2) Minimize the $n - 1$ conjectured machines and put them in the standard form. (3) Cross verify these machines until there is at most one left. (4) Confirm or reject the final one by applying a checking sequence.

[1] Conjecture

For a transition from s_j to s_{j+1} with input I_j, there are $n - 1$ possible changed next states s_r, $r \neq j + 1$. For each possibility, we have a modified specification machine $\tilde{A}_r$ where input I_j takes it from state s_j to s_r instead and outputs $\tilde{O}_j$. The modified machine $\tilde{A}_r$ is still deterministic and fully-specified, but may not be minimized. But we know its specification completely.

[2] Minimization

We minimize each conjectured machine from Step [1] and put it in the standard form [Ho71, Ko78]:

$$\tilde{A}_1, \cdots, \tilde{A}_{n-1} . \tag{3.5}$$

The reason we record them in the standard form is as follows. For two machines with no more than n states in this form, it

takes time $O(pn)$ to determine if they are isomorphic or not. It takes time $O(pn)$ to find an input sequence of length at most n that distinguishes the two machines. More specifically, if we apply the sequence to the two machines with both of them at their initial states then we shall observe different outputs. For details, see [Mo56, Ko78].

[3] **Cross Verification**

We now examine a pair of machines in (3.5): $(\tilde{A}_j, \tilde{A}_{j+1})$, $j = 1, \cdots, n-2$. If they are isomorphic, then we discard $\tilde{A}_j$ and consider the next pair $(\tilde{A}_{j+1}, \tilde{A}_{j+2})$. If they are not isomorphic then we construct a sequence that distinguishes the two machines. We apply this sequence to both machines and to B; we shall observe different output sequences y_j and y_{j+1} from the two machines $\tilde{A}_j$ and $\tilde{A}_{j+1}$, respectively. We also shall observe a corresponding output sequence y from B. Note that all three machines have to start at the initial state by a reset. If $y_j = y$ then we keep machine $\tilde{A}_j$ for further cross verification; $\tilde{A}_{j+1}$ and B have different outputs and can never be the isomorphic. For a same reason, if $y_{j+1} = y$ then we keep machine $\tilde{A}_{j+1}$. If neither of them is equal to y then we discard both of them; neither of them can be B. In any case, we discard at least one machine by a cross verification. We then take the next machine $\tilde{A}_{j+2}$ (or two) for further cross verification.

We finally stop with either none or one machine left. In the first case, none of the conjectured machines in (3.5) are isomorphic to B, and we know that the transition under consideration - the transition from state s_j to s_{j+1} upon input I_j - has not been changed, and we move on to process the next transition, which takes the machine from state s_{j+1} to s_{j+2} upon input I_{j+1}. In the second case, the transition under consideration may or may not be changed. We carry the leftover conjectured machine to pair with the conjectured machines from processing the next transition, and continue cross-verification.

[4] **Confirmation**

When we have considered all the k transitions of x_k, we have either one $\tilde{A}$ or no machine left. In the first case, we use $\tilde{A}$ as the specification and run a checking experiment on B. If B is isomorphic to $\tilde{A}$ then we have located the change in the implementation machine B. Thus, we have identified B, which is exactly $\tilde{A}$. Otherwise, $\tilde{A}$ is still a wrong conjecture. Since we have considered and ruled out all possible machines with a single change, we conclude that the implementation machine B has more than one change from the specification machine A.

There are $k < 2n$ transitions to check. For each transition, we construct $n-1$ possible changed machines, minimize them, and record them in the standard form. The total time is $O(pkn^2 \log n)$. To cross verify a pair of machines takes time $O(pn)$ and the total time for cross verification is $O(pkn^2)$. The confirmation in Step [4] takes time $O(pn^3)$. Therefore,

Theorem 1. Suppose that the machines under consideration have a reliable reset capability. If there is only a single change, then we can locate the change and identify the implementation machine B in time $O(pn^3 \log n)$ where p is the number of inputs and n is the number of states of the original specification machine A. If there are multiple changes, then we can still detect their presence with the same cost.

□

3.2.4. AN INCREMENTAL TECHNIQUE

The following technique when applied to the procedures given above would result in an improvement of an order of magnitude on the average. It runs Step [4] more often. In the worst case, the modified procedure has the same complexity as the original procedure.

Assume that an input sequence $x = I_1 \cdots I_k$ gives different outputs from A and B and we want to identify where is the changed transition. Instead of assuming that any of the $2n$ transitions are changed, we first assume that only the first log n transitions could be changed and then verify it. If all such conjectures are ruled out then we process the first $2 \log n$ transitions. We repeat the process, and each time we double the length of the transition sequences until it is of length k. The rationale is: with probability almost one an FSM has diameter log n [Kor67, TB73]. Therefore, if a sequence catches a change, it is very likely that it catches it within the first log n transitions.

3.3. EXAMPLE 1

We use an example to explain the fast algorithm given in Section 3.2.3. The specification FSM A is fully-specified with four states: s_1, s_2, s_3, and s_4, where s_1 is the initial state, four inputs: r, a, b, and c, where r is the reset input, and three outputs: x, y, and z. See Fig. 1.

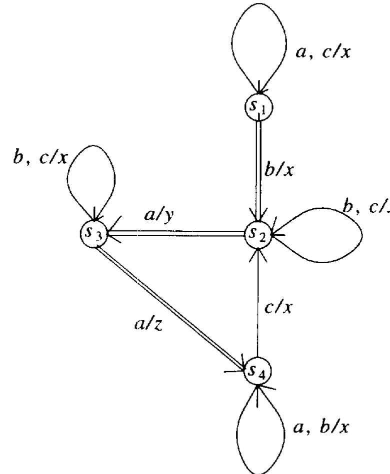

Fig. 1. Original specification machine A.

In Fig. 1, the label *a, c/x* means that application of input a or c would cause the state transition with an output x. Since the output changes can be easily identified, we only discuss next-state changes.

Suppose that we only have a single change: from state s_4

upon input a, the implementation machine B moves to state s_3 instead of staying in state s_4. See Fig. 2. This next-state change is to be identified from the I/O behavior of B.

Given two sets of states Q_1 and Q_2, a *separating sequence* is an input sequence x such that for each state pair $s_1 \in Q_1$ and $s_2 \in Q_2$, starting from the two states and applying x, different output sequences will be generated. The input sequence x separates states in Q_1 from those in Q_2. We use $Q_1 \mid Q_2$ to denote a separation of the states in Q_1 from those in Q_2.

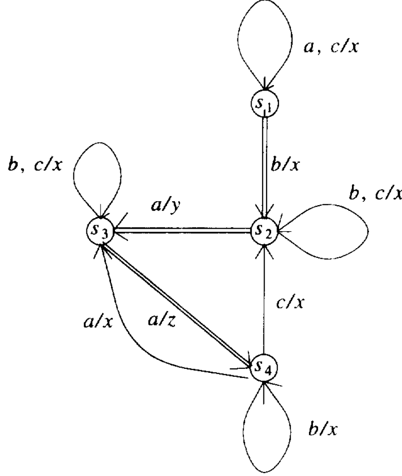

Fig. 2. The implementation machine B to be identified.

A characterization set of sequences for machine A is:

$$\begin{aligned} &a: \{s_1, s_4\} \mid \{s_2\} \mid \{s_3\}; \\ &ba: \{s_1\} \mid \{s_4\}. \end{aligned} \tag{3.6}$$

This means that the input a can distinguish between states in sets $\{s_1, s_4\}$, s_2, and s_3.

A checking sequence can be constructed as follows. Since the topic of this paper is not on checking experiments, we omit the details.

We check the states and tree edges (represented by doubled lines in Fig. 1) by:

$$\begin{aligned} &ra,\ rba\ (s_2 \mid s_1),\ rbaa\ (s_3 \mid \{s_1, s_2\}), \\ &rbaaa\ (s_4 \mid \{s_2, s_3\}), rba,\ rbaaba\ (s_4 \mid s_1). \end{aligned} \tag{3.7a}$$

We check each non-tree edge by first testing the transition and then its tail state:

$$\begin{aligned} &(s_1, a, c/x, s_1) : raa,\ raba,\ rca,\ rcba; \\ &(s_2, b, c/x, s_2) : rbba,\ rbca; \\ &(s_3, b, c/x, s_3) : rbaba,\ rbaca; \\ &(s_4, a, b/x, s_4) : rbaaaa,\ rbaaaba,\ rbaaba,\ rbaabba; \\ &(s_4, c/x, s_2) : rbaaca. \end{aligned} \tag{3.7b}$$

All the sequences in (3.7a) and (3.7b) consist of a basic test set of the checking sequence, interposed with reset r. Note that if a prefix of a sequence can detect a change (fault) then the whole sequence also can. Therefore, we can remove all the sequences that are prefix of other sequences, and we obtain the following basic test set which has 13 input sequences:

$$\begin{aligned} &raa,\ raba,\ rbaaaa(*),\ rbaaaba,\ rbaaba,\ rbaabba, \\ &rbaaca,\ rbaba,\ rbaca,\ rbba,\ rbca,\ rca,\ rcba \end{aligned} \tag{3.8}$$

Note that the ordering makes no difference, but we keep them in a lexicographic order.

We test them on B and the third one (with a *) catches a change; if we apply it to A, we have an output sequence *xyzxx*, and if we apply it to B, we have *xyzxz*. The fifth outputs are different. Even though the output discrepancy occurs at the fifth transition, the next-state error may occur at any of the five transitions. The test sequence we use for identification is

$$x_5 = baaaa. \tag{3.9}$$

We first conjecture that the first transition goes to a different state; instead of s_2, the machine goes to s_1, s_3, or s_4. Each conjecture assumes a new machine and we have three different machines to consider. We shall omit the details of this part and mention that *rba* rules out the first and the second possibility and that *rbba* rules out the third possibility. Therefore, the change cannot occur at the first transition. Similarly, we conclude that the change cannot occur at the second and third transitions. We now examine the fourth transition in detail.

The first three inputs *baa* take the machine to state s_4 and the fourth input a is supposed to take the machine to state s_4. A single next-state change can take the machine to state s_1, s_2, or s_3, instead, and gives a different machine $\tilde{A}_1$, $\tilde{A}_2$, and $\tilde{A}_3$, respectively. We construct these three machines and represent them in the standard form (they are minimized):

s	a	b	c
s_1	s_1, x	s_2, x	s_1, x
s_2	s_3, y	s_2, x	s_2, x
s_3	s_4, z	s_3, x	s_3, x
s_4	s_1, x	s_4, x	s_2, x

Table 1a. $\tilde{A}_1$.

s	a	b	c
s_1	s_1, x	s_2, x	s_1, x
s_2	s_3, y	s_2, x	s_2, x
s_3	s_4, z	s_3, x	s_3, x
s_4	s_2, x	s_4, x	s_2, x

Table 1b. $\tilde{A}_2$.

s	a	b	c
s_1	s_1 , x	s_2 , x	s_1 , x
s_2	s_3 , y	s_2 , x	s_2 , x
s_3	s_4 , z	s_3 , x	s_3 , x
s_4	s_3 , x	s_4 , x	s_2 , x

Table 1c. $\tilde{A}_3$.

We first consider the pair $(\tilde{A}_1 , \tilde{A}_2)$. Since both machines are in the standard form, one can easily find an input sequence that will produce different output sequences from the two machines starting from the initial state s_1. Such a sequence is *rbaaaa*, and the corresponding output sequences are as follows. From $\tilde{A}_1$: *xyzxx*; and from $\tilde{A}_2$: *xyzxy*. The corresponding output sequence from B is: *xyzxz*, which is different from both of them. We discard both of them. The only remaining machine now is $\tilde{A}_3$.

Next we have to check the fifth transition. Upon input a the machine moves from state s_4 to s_4 which is exactly the fourth transition. If there are no changes in the fourth transition, then there can be no changes in the fifth one. Therefore, we can skip checking the fifth transition.

Now we have only machine $\tilde{A}_3$ left. For the final confirmation, we find a checking sequence for $\tilde{A}_3$ in the same way as for A. We then run it on B and conclude that B is identical to $\tilde{A}_3$. We have identified the implementation machine B that is specified in Table 1c and Fig. 2. The single change occurs at state s_4; upon input a, the machine B moves to state s_3, instead of state s_4. Thus, we have reverse-engineered the implementation machine B.

4. A HEURISTIC PROCEDURE

A test sequence generated using the exact procedure in Section 3 can be very long. Here we give a heuristic procedure which is of the same length as a conformance test using the procedure in [SD88]. The test generation procedures given in [DSU90, SD88, ADLU91] have been widely used in practice because of the high fault coverage and relatively short lengths. Any test of comparable length for this reason may be acceptable for practical applications.

4.1. DETAILS OF THE PROCEDURE

Our heuristic procedure has four steps.

[1] Apply a test consisting of the following input sequences for each transition:

[r, guiding sequence, test edge, UIO sequence]

where r is the reset input, guiding sequence is the shortest path from the initial state to the test state; the test edge is the transition being tested; UIO sequence is the unique I/O sequence for the tail state.

[2] Collect the test results. From these test results, compute a syndrome consisting of a three-bit Boolean vector for each sequence: [x, y, z]

where $x = 0$ if the implementation generates expected outputs for the guiding sequence and $x = 1$ otherwise; $y = 0$ if the implementation generates expected outputs for the test edge and $y = 1$ otherwise; $z = 0$ if the implementation generates expected outputs for the UIO sequence and $z = 1$ otherwise.

The collection of these three-bit vectors will be called *syndrome*.

[3] Interpretation of the syndrome.

(1) Output Change:

In only one three-bit vector, y should be 1. Suppose that $y = 1$ for a test edge e_t, then any guiding sequence or UIO sequence which contains e_t should have the corresponding x or z equal to 1. If this is not true, then there must be a next-state change or multiple changes.

(2) Next-state Change:

In the three-bit vector syndromes, find one with $x = 1$. If this is the case, then there must be an edge (transition) in the shortest path to the test edge, which has a changed next-state. Find this edge from observing the syndromes of the transitions on the path. This results in a hypothesis of a possible next-state change, or, a conjectured machine.

If all x's are 0 then find the transition where the z-bit is 1, and analyze its corresponding UIO sequences for a possible next-state change. Similar to processing the x-bit, we find it by observing the syndromes of the transitions in the UIO sequence and obtain a conjectured machine for further verification.

[4] Verify the hypothesis by applying a checking sequence for the conjectured implementation machine obtained in the previous step.

Intuitively, $x = 1$ signals a possible next-state change on a path to the transition under test, $y = 1$ signals a possible output change, and $z = 1$ signals a possible next-state change of the transitions of the UIO sequence of the ending state of the transition under test. We "localize" the search for the change into three different categories. This will be further explained in Example 2.

4.2. EXAMPLE 2

The specification of a protocol entity is given in Fig. 3. The set of states $S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$ with s_1 as the initial state. The set of inputs is $I = \{r, a, c, x, z\}$, and the set of outputs is $O = \{null, b, d, f\}$. The output is *null* upon a reset input r. The inputs not shown in this figure are ignored by the machine. For such an input, we assume that the machine remains in its current state generating the *null* output.

The UIO sequences for each state are given in Table 2. First we will generate a test sequence for checking strong conformance [SD88].

The test set in Step 1 has 28 sequences (we are assuming reliable reset as discussed in Section 2). Let us consider a changed implementation B where the label of edge (s_3 , s_5) has been changed from *z/b* to *z/d*, see Fig. 3.

The 3-bit vectors for this changed implementation are given in Table 3. In these vectors, only one y for the state s_3 with input z is equal to zero. It is also from the examination of these bit vectors that the guiding sequences and UIO sequences which have this edge (s_3, s_5) also have the corresponding x's and y's equal to 1.

Let us consider another changed implementation with one next-state change shown in Fig. 4. The edge (s_2, s_3) with label *c/d* in the original specification is now changed to (s_2, s_5) with the same input and output. The three-bit vectors for this implementation are given in Table 4.

All x's for state s_5 are equal to 1. This means that there must be a problem in the guiding path from its initial state. Guiding paths to states s_2 and s_3 do not reveal any problems. But the UIO sequence for state s_3 reveals a discrepancy; this means that the transition from state s_2 to s_3 must have a changed next state. Only hypothesis that it now goes to state s_5 does not contradict all parts of the syndrome.

s	UIO
s_1	a/b, c/d
s_2	c/d
s_3	z/b, a/b
s_4	z/b, x/d
s_5	a/b, a/b
s_6	x/d
s_7	a/f, c/d

Table 2. *UIO sequences for the machine in Fig. 3.*

s	a	c	x	z
s_1	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)
s_2	(0,0,0)	(0,0,1)	(0,0,0)	(0,0,0)
s_3	(0,0,1)	(0,0,1)	(0,0,1)	(0,1,0)
s_4	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)
s_5	(1,0,0)	(1,0,0)	(1,0,0)	(1,0,0)
s_6	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)
s_7	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)

Table 3. Syndrome for implementation with one output change.

s	a	c	x	z
s_1	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)
s_2	(0,0,0)	(0,0,1)	(0,0,0)	(0,0,0)
s_3	(0,1,1)	(0,0,1)	(0,0,1)	(0,1,0)
s_4	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)
s_5	(1,0,0)	(1,0,0)	(1,0,0)	(1,0,0)
s_6	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)
s_7	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)

Table 4. Syndrome for implementation with one changed next-state transition.

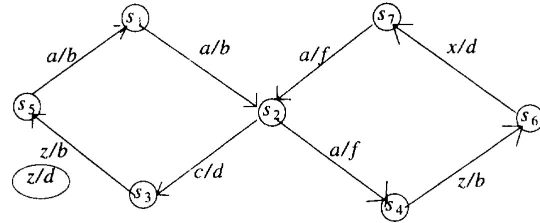

Fig. 3. Original specification machine A.

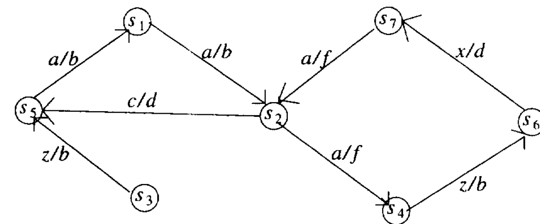

Fig. 4. The implementation machine B to be identified.

5. CONCLUSIONS

We have studied the problem of locating differences between a protocol specification and its implementation. We have solved it for the case of a single change. We give an exact procedure which has a complexity of $O(pn^3 \log n)$. We also give a heuristic procedure in which the identification sequence is comparable in length to a conformance test sequence given in [SD88].

As indicated earlier, if we could monitor the implementation machine frequently enough before multiple changes occur, then single-change assumption is reasonable. However, multiple changes still may happen in practice. For an arbitrary number of changes, it becomes a machine identification problem, which is known to have exponential complexity [Mo56]. However, if there are only a small number of changes, the problem is still manageable; if there are a constant number of next-state and output changes, then we have polynomial time algorithms for exact identification. More specifically, suppose that there are μ next-state changes and ν output changes, then with the same assumptions the implementation machine can be identified in time $O(p^{\mu+\nu} q^{\nu} n^{2\mu+\nu})$ where q is the number of outputs. This will be reported in the full paper [LS93].

The work reported here is just the beginning. For example, practical heuristic procedures have to be designed for the case of multiple changes. This problem has also to be solved for

other protocol models such as communicating FSM's and extended FSM's. An important question is whether the reverse engineering problem can be simplified by increased observability of implementation state.

ACKNOWLEDGEMENT

We thank David Kristol, Raymond E. Miller, and Hasan Ural for their valuable comments.

REFERENCES

[ADLU91]A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar [1991], "An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours," in *IEEE Trans. on Communication*, Vol. 39, No. 11, November 1991.

[An92]D. Angluin [1992], "Computational Learning Theory: Survey and Selected Bibliography," in *Proc. of the 24th Annual ACM Symp. on The Theory of Computing*, pp. 351-369.

[Ch78]T. S. Chow [1978], "Testing Software Design Modeled by Finite-State Machines," in *IEEE Trans. on Software Engineering*, Vol. SE-4, No. 3, pp. 178-187.

[DSU90]A. T. Dahbura, K. Sabnani, and M. U. Uyar, "Formal Methods for generating Protocol Conformance Test Sequences," *Proceedings of the IEEE*, Vol. 78, No. 8, August 1990.

[GB92]A. Ghedamsi and G. v. Bochmann [1992], "Test Result Analysis and Diagnostics for Finite State Machines," in *Proc. of the 12th Int. Conf. on Distributed Systems*, Yokohama, Japan.

[GBD93]A. Ghedamsi, G. v. Bochmann, and R. Dssouli [1993], "Multiple Fault Diagnostics for Finite State Machines," in *Proc. INFOCOM 93*, pp. 782-791.

[He64]F. C. Hennie [1964], "Fault Detecting Experiments for Sequential Circuits," in *Proc. Fifth Ann. Symp. Switching Circuit Theory and Logical Design*, pp. 95-110, Princeton, N. J.

[Ho71]J. E. Hopcroft [1971], "An n log n Algorithm for Minimizing States in a Finite Automaton," in *Theory of Machines and Computations*, Z. Kohavi and A. Paz (eds.), pp. 189-196, Academic Press, New York.

[Ko78]Z. Kohavi [1978], *Switching and Finite Automata Theory*, 2nd Edition, McGraw-Hill Book company, New York.

[KK68]I. Kohavi and Z. Kohavi [1968], "Variable-Length Distinguishing Sequences and Their Application to the Design of Fault-Detection Experiments," in *IEEE Trans. on Computers*, Vol. C-17, pp. 792-795.

[Kor67]A. D. Korshunov [1967], "On the Degree of Distinguishability of Finite Automata," in *Diskretnyi Analiz.*, pp. 39-59.

[LS93]D. Lee and K. Sabnani [1993], "Reverse-engineering of Communication Protocols," in preparation.

[LY93]D. Lee and M. Yannakakis [1993], "Testing Finite State Machines: State Identification and Verification," to appear in *IEEE Trans. on Computers*.

[ML90]R. E. Miller and G. M. Lundy [1990], "Testing Protocol Implementations based on a Formal specification," *Protocol Test Systems III*, North Holland, 1991, pp. 289-304.

[MP91]R. E. Miller and S. Paul [1991], "Generating Minimal Length Test Sequences for Conformance Testing of Communication Protocols," in *Proc. INFOCOM 91*.

[Mo56]E. F. Moore [1956], "Gedanken-experiments on Sequential Machines," in *Automata Studies*, Annals of Mathematics Studies, No. 34, pp. 129-153, Princeton University Press, Princeton, N. J.

[SD88]K. K. Sabnani and A. T. Dahbura [1988], "A Protocol Test Generation Procedure," in *Computer Networks and ISDN Systems*, Vol. 15, No. 4, pp. 285-297.

[TB73]B. A. Trakhtenbrot and Y. M. Barzdin [1973], *Finite Automata, Behavior and Synthesis*, North-Holland Pub. Co., Amsterdam.

[UP83]H. Ural and R. L. Probert [1983], "User-guided test sequence generation," *Protocol Specification, Testing, and Verification, III*, North Holland, eds. H. Rudin and C. H. West, 1983, pp. 421-436.

[Va84]L. G. Valiant [1984], "A Theory of Learnable,'' in *C. ACM*, **27**, pp. 1134-1142.

[Vas73]M. P. Vasilevskii [1973], "Failure Diagnosis of Automata," in *Kibernetika*, No. 4. pp. 98-108.

[WL92]C.-J. Wang and M. T. Liu [1992], "A Test Suite Generation Method for Extended Finite State Machines using Axiomatic Semantics Approach," *Protocol Specification, Testing, and Verification, XII* North Holland, ed. M. U. Uyar and J. Linn, 1992.

[YL93]M. Yannakakis and D. Lee [1993], "Testing Finite State Machines: Fault Detection," to appear in *J. of Computer and System Sciences*.