Efficient Computation of Unique Input/Output Sequences in Finite-State Machines

Kshirasagar Naik, *Member, IEEE*

***Abstract*—This paper makes two contributions toward computing unique input/output (UIO) sequences in finite-state machines. Our first contribution is to compute *all* UIO sequences of minimal lengths in a finite-state machine. Our second contribution is to present a *generally efficient* algorithm to compute a UIO sequence for each state, if it exists. We begin by defining a *path vector, vector perturbation,* and *UIO tree*. The perturbation process allows us to construct the complete UIO tree for a machine. Each sequence of input/output from the initial vector of a UIO tree to a singleton vector represents a UIO sequence. Next, we define the idea of an *inference rule* that allows us to infer UIO sequences of a number of states from the UIO sequence of some state. That is, for a large class of machines, it is possible to compute UIO sequences for all possible states from a small set of initial UIO's. Thus, there is neither any need for individually computing UIO sequences nor any need to construct the complete UIO tree. We give a modified *depth-first* algorithm, called the *hybrid* approach, that computes a partial UIO tree, called an *essential subtree,* from which UIO sequences of all possible states can be inferred. Using the concept of *projection machines,* we show that sometimes it is unnecessary to construct even a partial subtree. We prove that if a machine remains strongly connected after deleting all the converging transitions, then all of the states have UIO sequences. To demonstrate the effectiveness of our approach, we develop a tool to perform experiments using both small and large machines. Experimental results show that the maximum essential subtree for a machine is much smaller than the complete UIO tree. An immediate impact of the inference mechanism is that UIO sequences may be unnecessarily long leading to longer test sequences. Since longer test sequences incur extra cost, we suggest how to reduce—and even eliminate—the impact of the inference mechanism by generating UIO sequences depending on how they are used in some representative test generation methods.**

***Index Terms*— Communication protocol, finite-state machine, interference rule, path vector, projection, testing, UIO sequence, UIO tree.**

I. INTRODUCTION

FINITE-STATE machines (FSM's) have been found to be very useful in modeling both hardware, such as sequential circuits, and software, such as parsing an input program to a compiler. FSM's are also useful in modeling the control portion of data communication protocols. From the point of obtaining implementations of communication protocols, FSM models seem to be more popular than other specification techniques such as temporal logic, Petri Net, and process algebra. Testing an implementation of an FSM in the "black box" approach remains to be a challenging task, as will be explained in detail.

A. FSM and Graph-Theoretic Terms

A finite-state machine M is a tuple $M = (S, I, O, \delta, \lambda)$, where $S = \{s_1, \cdots, s_n\}$ is a finite set of states, $I = \{a_1, \cdots, a_p\}$ is a finite set of inputs, $O = \{b_1, \cdots, b_r\}$ is a finite set of outputs, δ: $S \times I \rightarrow S$ is the state transition function, and λ: $S \times I \rightarrow O$ is the output function. For convenience, we extend the domain of λ and δ from an input symbol to a string of symbols. For instance, for state s_i and input sequence $x = a_1, \cdots, a_k$, the corresponding output sequence is denoted by $\lambda(s_0, x)$ and the final state is denoted by $\delta(s_0, x)$. That is, $\lambda(s_0, x) = b_1, \cdots, b_k$, where $b_i = \lambda(s_{i-1}, a_i)$ and $s_i = \delta(s_{i-1}, a_i)$ for $i = 1, \cdots, k$, and the final state is $\delta(s_0, x) = s_k$. Also, we extend the transition and output functions from single states to sets of states. That is, if Q is a set of states and x is an input sequence, then $\delta(Q, x) = \{\delta(s, x) | s \in Q\}$, and $\lambda(Q, x) = \{\lambda(s, x) | s \in Q\}$.

An FSM is said to be *reduced* if for any pair of states s_i and s_j, $i \neq j$, there is an input sequence y, such that $\lambda(s_i, y) \neq \lambda(s_j, y)$. Intuitively, an FSM is reduced if no two states are equivalent. That is, there exists a sequence of input that distinguishes one state from the other. An FSM is said to be *deterministic* if for each input $a \in I$, there is at most one transition defined at each state of M. An FSM is said to be *completely specified* if for each input $a \in I$, there is a transition defined at each state of M.

An FSM M is viewed as a directed graph $G = (V, E)$, where the set of vertices $V = \{v_1, \cdots, v_n\}$ represents the set S of states of M, and the set of edges $E = \{(v_j, v_k; a/b) | v_j, v_k \in V\}$ represents all transitions in M. Each edge $e_{jk} = (v_j, v_k; a/b) \in E$ represents a state transition from s_j to s_k with input a and output b. Notationally, v_j, v_k, and a/b are called the *head, tail,* and *label* of e_{jk}, and are denoted by $head(e_{jk})$, $tail(e_{jk})$, and $IO(v_j, v_k)$, respectively.

We denote the *in-degree* and *out-degree* of a vertex v by $in(v)$ and $out(v)$, respectively. In any graph, vertex v_i with $in(v_i) = 0$ is called a *source* vertex, and v_j with $out(v_j) = 0$ is called a *sink* vertex. Notation $x@y$ denotes the *concatenation* of strings x and y. In the following, G and M, vertices and states, and edges and transitions are used interchangeably.

B. Protocol Testing at Large

FSM-based protocol testing has been widely investigated during the last few years [4]–[6], [8]–[12]. The three major components of a testing strategy are:

Manuscript received February 8, 1995; revised December 1996; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor K. Sabnani.

The author is with the Department of Computer Software, University of Aizu, Aizu-Wakamatsu, Japan 965 (e-mail: k-naik@u-aizu.ac.jp).

Publisher Item Identifier S 1063-6692(97)05580-5.

1) a set of assumptions about the FSM specification;
2) a set of assumptions about an implementation;
3) a test generation methodology.

In an FSM-based testing strategy, it is generally assumed that the FSM model of a protocol is *completely specified, deterministic, reduced,* and *strongly connected.* We call a machine satisfying these four assumptions a *CDRS machine.* Some strategies do not assume that the machine be completely specified [8]–[10]. About an implementation, it is assumed that the implementation does not have more states than the specification model. Given the state table of a machine M and its implementation M', satisfying their respective assumptions, following are the goals of a test strategy for s_i and a_i, for all i.

1) *Homing:* Move M' to state s.
2) *Output Verification:* Apply input a and verify that M' outputs $\lambda(s, a)$.
3) *State Verification:* Verify that M' moves to next state $\delta(s, a)$.

The first step is known as *homing* a machine to a desired state, and it is known that a preset homing sequence, whose length is at most $(n-1)^2$, exists for every reduced n-state machine [3]. The second step verifies whether M' produces the desired output associated with a state transition. Output verification is a trivial one assuming that the machine produces an output after an arbitrary but finite delay of receiving an input. The third step is known as state verification: verify that M' is in the expected state $s' = \delta(s, a)$. In the following subsection, we discuss the state verification problem in detail.

C. State Verification and Computational Complexity

The two kinds of sequences widely used in state verification in a machine M are: *distinguishing* sequence (D sequence) and *unique input/output* (UIO) sequence.

Input sequence x is said to be a *preset* D sequence iff $\lambda(s_i, x) \neq \lambda(s_j, x)$, $i \neq j$, for all state pairs s_i, s_j in M. In an *adaptive* D sequence, the kth input depends on the output corresponding to the $(k-1)$th input. In fact, an adaptive D sequence is a tree of inputs and outputs, rather than a sequence of inputs [3].

Input/output sequence $y/\lambda(s_i, y)$ is said to be a UIO sequence for state s_i iff $y/\lambda(s_i, y) \neq y/\lambda(s_j, y)$, $i \neq j$, $\forall s_j$ in M. Notationally, $\text{UIO}(s_i) = y/\lambda(s_i, y)$.

Table I summarizes the relationship between D sequence and UIO sequence, and Table II summarizes the complexity of computing those sequences in an arbitrary CDRS machine. Table II suggests that state verification using a preset D sequence is computationally prohibitive. In spite of the difficulty in designing adaptive experiments [3], state verification using an adaptive D sequence looks very practical. However, not all CDRS machines possess adaptive D sequences. On the other hand, there are machines which do not have adaptive D sequences, but do have UIO sequences for some or all states [12]. Thus, the class of machines whose states can be verified using UIO sequences is larger than the class of machines whose states can be verified using D sequences.

The only algorithm for direct computation of UIO sequences, referred to as the SD algorithm in the rest of this paper, is due to Sabnani and Dahbura [4]. The time complexity of the algorithm is shown in Table II.

TABLE I
RELATIONSHIP BETWEEN D SEQUENCE AND UIO SEQUENCE

$\Longrightarrow$	Preset D-seq.	Adap. D-seq.	UIO Seq. (All states)
Preset D-seq.	(Trivial)	Always	Always
Adaptive D-seq.	Sometimes	(Trivial)	Always
UIO (All states)	Sometimes	Sometimes	(Trivial)
UIO (Not All states)	Never	Never	(Trivial)

TABLE II
EXISTENCE AND COMPUTATIONAL COMPLEXITY OF D SEQUENCE AND UIO SEQUENCE

	Existence	Time Complexity
Preset D-seq.	Not all machines	$O((n-1)n^n)$ [3]
Adaptive D-seq.	Not all machines	$O(pn\ log(n))$ [12]
UIO Seq.	Some states may not	$O(n^2(d_{max})^{(2n^2+2)})$ [4]

p = number of inputs, n = number of states, d_{max} = largest out-degree for any state.

Here, we analyze the central idea in the SD algorithm. This analysis will help us in understanding the basic difference between the SD algorithm and our algorithm. Let us study the following informal description of the SD algorithm:

> For each state, all input/output sequences of length 1 are computed and checked for uniqueness. If there is no unique sequence of length 1, then the same procedure is repeated for all sequences of length 2. This procedure is continued for longer sequences until a unique input/output sequence is found or the length of sequence exceeds $2n^2$.

The following are three key observations in the above description.

1) The SD algorithm builds the search space to compute UIO sequences in a *breadth-first* manner.
2) All of the UIO sequences are *independently* constructed from scratch. That is, the algorithm starts with sequences of length 1, and incrementally builds longer sequences.
3) The algorithm terminates when a UIO sequence for each state is found or the sequence length exceeds $2n^2$, whichever occurs earlier. Thus, the SD algorithm does not compute UIO sequences of longer lengths.

In the following subsection, we state some useful observations on the structure of a machine. These observations lead us to design a new algorithm whose average case performance is much better than the SD algorithm.

D. Central Idea in This Paper

Our first claim is that, by defining a suitable information structure for UIO construction, it is possible to compute all UIO sequences of minimal lengths if they exist. We can define a termination condition for the algorithm that depends on the machine structure, rather than on $2n^2$ as is the case in the SD algorithm.

Our second claim is that all UIO sequences need not be computed from scratch. Rather, if we know a UIO sequence for one state, we can construct UIO sequences for several other states in polynomial time. Thus, by constructing a part of the search space, we can functionally substitute the rest of the search space of exponential size by a polynomial process. On an average, this results in significant reduction in search space.

Our third claim is that, in order to gain significant benefit from our second claim, a modified *depth-first*, called *hybrid*, approach to UIO construction is better suited than the breadth-first approach.

In the following subsection, we give a detailed outline of our contribution.

E. Our Contribution

First, we define the concept of a *path vector* and the idea of *vector perturbation* using an input/output edge label. Combining these two ideas, we define a *UIO tree*. A useful property of a UIO tree is that the sequence of input/output from the initial node of the UIO tree to a singleton terminal node is a UIO sequence. Additionally, the UIO tree construction algorithm is both *sound* and *complete*.

Second, we define an *inference rule* that allows us to infer UIO sequences for several states from a UIO sequence of some state and the transition relations of a machine. Therefore, by computing a few UIO sequences in a kind of *depth-first* manner, we can infer UIO sequences for all other states, if they exist. Thus, largely there is no need to construct the complete UIO tree for a machine.

Third, we introduce the idea of *projection machines* to possibly compute UIO sequences for some states in polynomial time without constructing any UIO tree. In case the projection technique fails to yield any initial UIO sequences, we resort to UIO tree construction to obtain some initial UIO sequences.

Fourth, we enhance our UIO tree construction algorithm so that it can operate in a kind of modified *depth-first* approach, called the *hybrid* approach. The central idea in this algorithm is that it searches for UIO sequences in a largely depth-first manner. When a UIO sequence for a state is found, it applies the inference rules to compute UIO sequences for additional states. The algorithm terminates when a UIO sequence is found for every state or the complete UIO tree is constructed.

Thus, the algorithm constructs the complete UIO tree in the worst case. However, by applying the algorithm to a number of machines, we find that, on an average, the size of the UIO tree constructed by the algorithm is much smaller than the full size.

Fifth, we prove a very useful relation between machines with *converging transitions* [10] and the existence of UIO sequences in those machines. In simple words, if a machine remains strongly connected after deleting all of the converging transitions, then each state of the machine has a UIO sequence.

The main disadvantage of the inference mechanism is that UIO sequences get longer and longer as we derive more and more of them. Since UIO sequences are used in the generation of test sequences, the impact of the inference mechanism on test sequences will be an increased test sequence length. Depending on the interpretation of the costs of executing transitions of a state machine, the extra length may be very much undesirable. Therefore, we propose two methods for reducing the impact of the inference mechanism, depending on how UIO sequences are used in various test generation techniques [8], [10].

In Section II, an algorithm, based on the idea of *path-vector perturbation*, is presented for direct computation of UIO sequences. The concept of inference rule is presented in Section III. Properties of projection machines are discussed in Section IV. Combining the idea of projection and inference, we present an algorithm for UIO construction in Section V. The relation between machines with converging transitions and the existence of UIO sequences is studied in Section VI. In Section VII, we develop a tool to study the performance of our approach. In Section VIII, we study two methods for reducing the impact of the inference mechanism on test sequence length. Finally, some concluding remarks are given in Section IX.

II. Path Vector and UIO Tree

In this section, we first define a *path vector* and *perturbation* of a path vector. Based on the perturbation idea, next we define a *UIO tree*. We incorporate two termination conditions into the perturbation process so that the algorithm terminates after generating all UIO sequences of minimal lengths. The termination conditions ensure the soundness and completeness of the algorithm.

Definition: Given an FSM M, a *path vector* (PV) is a collection of state pairs $(v_1/v_1', \cdots, v_i/v_i', \cdots, v_k/v_k')$ with the following properties:

1) v_i and v_i' denote the head and tail state, respectively, of a path, where a path is a sequence of state transitions;
2) an identical sequence of input/output is associated with all of the paths in the path vector.

Given a path vector PV $= (v_1/v_1', \cdots, v_i/v_i', \cdots, v_k/v_k')$, the *initial vector* (IV) is the collection of head states of PV, that is IV(PV) = $(v_1, \cdots, v_i, \cdots, v_k)$. Similarly, the *current vector* (CV) is the collection of tail states of PV, that is, CV(PV) = $(v_1', \cdots, v_i', \cdots, v_k')$. A path vector is said to be a *singleton* vector if it contains exactly one state pair. A path vector PV $= (v_1/v_1', \cdots, v_i/v_i', \cdots, v_k/v_k')$ is said be a *homogeneous* vector if all members of CV(PV) are identical. It may be noted that a singleton vector is also a homogeneous vector.

For an n-state machine, we define a unique *initial* path vector $(s_1/s_1, \cdots, s_i/s_i, \cdots, s_n/s_n)$ such that a *null* path is associated with all state pairs.

Now, we introduce the idea of *vector perturbation*, that is, given a path vector PV and an edge label a/b, how to compute a new vector PV$'$ from PV and a/b.

Definition: Given a PV $= (v_1/v_1', \cdots, v_i/v_i', \cdots, v_k/v_k')$ and an edge label a/b, *perturbation* of PV with respect to edge label a/b, denoted by PV$' =$ pert(PV, a/b), is defined as

$$\text{PV}' = \{v_i/v_i'' | v_i'' = \delta(v_i', a) \wedge \lambda(v_i', a) = b \wedge v_i/v_i' \in \text{PV}\}.$$

Given a reduced machine and its initial path vector, we can infinitely perturb all the path vectors for all edge labels. One can imagine the perturbation function PV$' =$ pert(PV, a/b) as an arc from a node PV to a new node PV$'$ with edge label a/b. Also, given a PV and a set of edge labels L, we can arrange the new $|L|$ nodes pert(PV, a/b), $\forall\, a/b \in L$ on one level. That is, all of the path vectors of a given machine can be arranged in the form of a tree with successive levels $1, 2, \cdots, \infty$. Such a tree is called a *UIO tree*.

Note: In the graphical representation of a path vector PV $= (v_1/v_1', \cdots, v_i/v_i', \cdots, v_k/v_k')$ in a UIO tree, we represent PV in two rows of states. The top row denotes IV(PV) and the bottom row denotes CV(PV).

Theoretically, a UIO tree is a tree with infinite levels. However, we need to prune the tree based on some conditions, called *pruning* conditions. After each perturbation PV$'$ = pert(PV, a/b), we check the following pruning conditions.

C1: CV(PV$'$) is a homogeneous vector.

C2: On the path from the initial node to PV, there exists PV$''$ such that PV$' \subseteq$ PV$''$.

If one of the pruning conditions is satisfied, we declare PV$'$ to be a *terminal* node. Now, we justify why C1 and C2 are two good termination conditions.

Let us consider condition C1. If CV(PV$'$) is a homogeneous vector, then CV(PV$'$) contains either exactly one element or identical elements. In case CV(PV$'$) contains exactly one element, we have found a UIO sequence for a state, and there is no need to further expand the UIO sequence. On the other hand, if CV(PV$'$) contains identical elements, then any further perturbation of the path from the initial node to PV$'$ will not lead to any UIO sequence. Let us consider condition C2. Condition C2 states that there is no need to perturb a node if the node or a superset of the node appears before. This is because, if the node or its superset appears before, then it has already been perturbed in all possible ways, and perturbing it once again will not lead to new information.

In the following, we present an algorithm to compute a finite UIO tree for a machine. In the rest of the paper, we use the terms *path vector* and *node* in a UIO tree interchangeably.

Algorithm 1: Generation of UIO tree.

Input: $M = (S, I, O, \delta, \lambda)$ and L.

Output: UIO tree.

Method: Execute the following steps.

Step 1: Let Ψ be the set of nodes denoting path vectors in the UIO tree. Initially, Ψ contains the initial vector marked as *nonterminal*.

Step 2: Find a nonterminal member $\psi \in \Psi$ which has not been perturbed. If no such member exists, then the algorithm terminates.

Step 3: Compute $\psi' =$ pert(ψ, a_i/b_i) and add ψ' to $\Psi \forall\, a_i/b_i \in L$. Mark ψ to be perturbed. Update the UIO tree.

Step 4: If pert(ψ, a_i/b_i), computed in Step 3, satisfies condition C1 or C2, then mark pert(ψ, a_i/b_i) as a *terminal* node and go to Step 2.

End of Algorithm 1

Remark 1: Since Algorithm 1 generates all UIO sequences of minimal lengths, it generates all possible *multiple* UIO sequences [9] for a state.

Example 1: The UIO tree for machine $G1$ in Fig. 1(a) is shown in Fig. 1(b). Here, we remind the reader that in the graphical representation of a UIO tree, we represent a node ψ in two rows of states: the top row represents IV(ψ), whereas the bottom row represents CV(ψ). The set of distinct edge labels is given by $L = \{0/0, 0/1, 1/0\}$. The initial node is given by $\psi_1 = (A/A, B/B, C/C, D/D)$. Initial node ψ_1 can be perturbed using all members of L as follows:

$$\begin{aligned}(A/B, B/A) &= \text{pert}(\psi_1, 0/0)\\ (C/D, D/D) &= \text{pert}(\psi_1, 0/1)\\ (A/D, B/B, C/A, D/C) &= \text{pert}(\psi_1, 1/0).\end{aligned}$$

TABLE III
ALL MINIMAL LENGTH UIO SEQUENCES IN MACHINE $G1$

A	$010/\lambda(A, 010)$
B	$010/\lambda(B, 010)$, $1010/\lambda(B, 1010)$, $11010/\lambda(B, 11010)$
C	$1010/\lambda(C, 1010)$
D	$11010/\lambda(D, 11010)$

Theorem 1: Given a machine $G = (V, E)$, state v has a UIO sequence iff the UIO tree has a singleton node ψ such that $v =$ IV(ψ).

Proof: (if part) Assume that the UIO tree has a singleton node ψ such that $v =$ IV(ψ). The sequence of input/output on the path from the initial node to ψ is a UIO sequence for state v because of the following two reasons:

1) A singleton node contains only one state pair denoting the head state and tail state of a path.
2) The perturbation process ensures that the sequence of input/output on a path from the initial node to any node is unique.

(only-if part) Assume that state v has a UIO sequence. Let UIO(v) be the sequence $a_1/b_1, \cdots a_i/b_i, \cdots a_k/b_k$ of minimal length. We consider minimal length UIO sequences because the algorithm does not perturb singleton vectors. We need to prove that UIO(v) leads to a singleton vector in the UIO tree. The proof is done by contradiction. Assume that the sequence UIO(v) leads to a terminal node with at least two components. For simplicity, we assume that it leads to a node with exactly two components: $(v/v', v_1/v_1')$. Thus, there exists a path from state v to v' with input/output label UIO(v), and there exists a second path from v_1 to v_1' with the same input/output sequence UIO(v). In that case, sequence UIO(v) cannot be a UIO sequence for state v. Thus, state pair v_1/v_1' does not appear with pair v/v' in any terminal vector ψ such that $v/v' \in \psi$, and input/output sequence UIO(v) is associated with the path from the initial node to ψ. Therefore, v/v' must appear as a singleton vector. □

Remark 2: The minimality of UIO sequences follows from the fact that singleton nodes are never perturbed.

Example 2: Referring to the UIO tree of Fig. 1 for machine $G1$, we list all minimal length UIO and multiple UIO sequences in Table III.

Remark 3: A UIO tree looks somewhat similar to a splitting tree constructed by the algorithm of Lee and Yannakakis [12] to derive an adaptive distinguishing sequence from an FSM. Therefore, it is interesting to compare the two kinds of trees. We expect some similarities between the two trees because an adaptive distinguishing sequence[1] is also a collection of UIO sequences for all states of an FSM. We also expect some

[1]It may be more appropriate to call an adaptive distinguishing sequence an *adaptive distinguishing tree*. Nevertheless, we use the phrase adaptive distinguishing sequence.

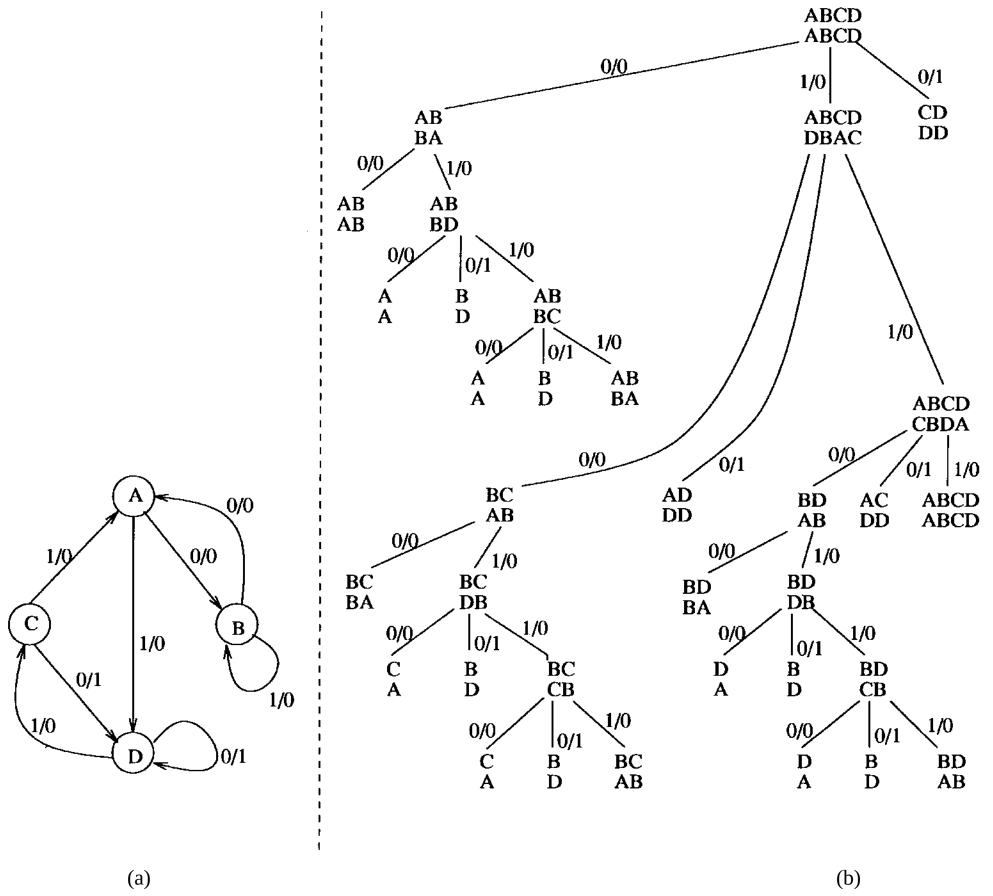

Fig. 1. (a) Machine $G1$. (b) UIO tree for Machine $G1$.

dissimilarities between the two trees because an FSM without an adaptive distinguishing sequence may have UIO sequences for some or all of its states. The two algorithms—ours and Lee and Yannakakis's—are compared in the following:

- Both of the algorithms split a node to obtain smaller nodes.
- The splitting tree algorithm obtains a *partition* of the set of states labeling a node, whereas our algorithm constructs possibly overlapping subblocks from the IV component of a node.
- Nodes of a splitting tree are partitioned using *three* types of *valid* inputs for a node, whereas we perturb a node with respect to a set of input/output pairs applicable to the IV component of a node.
- All leaf nodes of a splitting tree are singletons iff the FSM has an adaptive distinguishing sequence, whereas a singleton leaf of a UIO tree represents the existence of a UIO sequence for one state.
- Construction of a splitting tree is a "quick jump" process in the sense that, in partitioning a node, information about the already partitioned nodes is reused. This makes the height of the splitting tree much smaller than the length of the adaptive distinguishing sequence. On the other hand, the UIO tree construction process does not use any such information. This leads to the fact that the length of a UIO sequence is the same as the number of perturbations used to reach the corresponding singleton leaf.

III. Inference Rule

Informally, an *inference rule* is explained as follows. Assume that somehow we compute UIO(s_i) for state s_i. Then, we can infer a UIO sequence for another state s_j as follows:

$$\text{UIO}(s_j) = \text{IO}(s_j, s_i)@\text{UIO}(s_i),$$

where IO(s_j, s_i) is an edge label with some desired property to be explained soon. In fact, from one known UIO sequence, we can possibly infer several UIO sequences. Since a machine is assumed to be strongly connected, we can compute UIO sequences for all states, if they exist, from only a few initially known UIO sequences. Thus, there is no need to individually construct UIO sequences for all possible states.

The structure of the inference rule suggests that we can construct a database, called a *rule base*, consisting of all instances of the inference rule in a machine. Informally, the rule base has a size of order $O(pn)$, where p is the number of inputs and n is the number of states in a machine. The basic strategy is to avoid constructing the complete UIO tree for a machine. We construct only a small subtree of the complete UIO tree until we detect a few UIO sequences. Next, we use the rule base to compute UIO sequences for other states. Thus, in general, the inference rules, running in polynomial time complexity, render a large part of the UIO tree of exponential size to be redundant. An interesting observation is that even random computation of some initial UIO sequences results in significant saving in computation. In the following, we formalize the inference rule.

Definition: For state s_i, state s_j is said to be a *unique predecessor* if IO(s_j, s_i), that is, the label of edge (s_j, s_i), is unique among all incoming edges to state s_i. We denote the set of all unique predecessors of s_i by $S^u(s_i)$. State s_i is not included in $S^u(s_i)$ because we do not gain anything.

Example 3: Referring to machine $G1$ in Fig. 1(a), the sets of unique predecessors are as follows: $S^u(A) = \{B, C\}$, $S^u(B) = \{A\}$, $S^u(C) = \{D\}$, and $S^u(D) = \{A\}$.

Inference Rule: Let s_i be a state having a UIO sequence UIO(s_i), and let the set of unique predecessors of s_i be given by $S^u(s_i)$. We can infer a UIO sequence for state $s_j \in S^u(s_i)$ using the following relation:

$$\text{UIO}(s_j) = \text{IO}(s_j, s_i)@\text{UIO}(s_i)$$

where IO(s_j, s_i) is a unique label among all edges whose tail state is s_i.

Proof: We prove, by contradiction, that IO(s_j, s_i)@UIO(s_i) is a UIO sequence for state s_j. Assume that IO(s_j, s_i)@UIO(s_i) is not a UIO sequence for state s_j. Therefore, there exists another state s_k with IO behavior IO(s_j, s_i)@UIO(s_i). Hence, there exists a state s'_k such that IO(s_k, s'_k) = IO(s_j, s_i) and state s'_k has an IO sequence identical to UIO(s_i). Since the IO sequence UIO(s_i) is a UIO for state s_i, state s'_k cannot have the same IO behavior. Hence, no state other than s_j has IO behavior equal to IO(s_j, s_i)@UIO(s_i). □

A *rule base* is a collection of all instances of the inference rule. By referring to the set of unique predecessors for each state, we can easily construct a rule base for a machine.

Example 4: The following is a rule base for machine $G1$:

$$\begin{aligned}\text{UIO}(B) &= 0/0@\text{UIO}(A)\\ \text{UIO}(C) &= 1/0@\text{UIO}(A)\\ \text{UIO}(A) &= 0/0@\text{UIO}(B)\\ \text{UIO}(D) &= 1/0@\text{UIO}(C)\\ \text{UIO}(A) &= 1/0@\text{UIO}(D).\end{aligned}$$

Complexity of Using Inference Rules: In order to decide whether state s_j, for edge (s_j, s_i), belongs to $S^u(s_i)$, we must compare edge label IO(s_j, s_i) with all IO(s_k, s_i), where (s_k, s_i) is an edge in the graph. Thus, if there are $|E|$ edges in the graph, then the worst case complexity is given by $O(|E|^2)$. However, the average case complexity will be much less given that each edge label is not compared with all other edge labels. Rather, the label of each edge e_i is compared with $|E_i|$ edge labels, where $E_i \subseteq E$ and $\Sigma E_i = E$.

Now, we introduce the concept of an *inference* graph.

Definition: For a machine G, we construct an *inference graph* IG(G) as follows:

1) Represent each state s_i in G as a vertex in IG(G).
2) Put a directed edge from s_j to s_i iff $s_j \in S^u(s_i)$. Edge (s_j, s_i) is labeled with the unique IO(s_j, s_i) present in G.

Example 5: The inference graph for machine $G1$ is shown in Fig. 2.

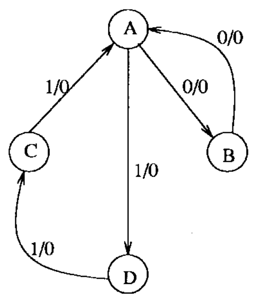

Fig. 2. Inference graph for machine $G1$.

Properties of an Inference Graph: The following are some properties of an inference graph.

1) For any directed path from state s_j to s_k, if we can compute a UIO sequence for s_k, then we can readily infer UIO sequences for all other states on the path.
2) For any cycle, if we find a UIO sequence for one state, we can infer UIO sequences for all other states on the cycle. Thus, either all states on a cycle have UIO sequences or none has UIO sequences.

We provide some additional remarks on inference graphs in Section VI.

Comments on Initial UIO Computation: Using the concept of inference rules, we argued that it is unnecessary to directly construct UIO sequences for all states. If we know the UIO's for some states, called initial UIO's, then we can infer UIO's for the rest of states. However, so far we have not discussed how to compute the initial UIO's. There is a need to efficiently compute the initial UIO's so that the benefit from using the inference rules is significant.

One immediate question is whether we can use the SD algorithm or our Algorithm 1 to compute some initial UIO sequences, so that using the inference rules results in better complexity. In order to decide their suitability, we briefly review the two algorithms in the following.

The SD algorithm works in a *breadth-first* manner. That is, the algorithm computes all sequences of length 1, 2, and so on. Thus, in case the UIO sequences are of almost equal length, using the inference rules will not be very effective in reducing computation. Thus, the breadth-first approach does not seem to be appropriate in computing some initial UIO sequences. On the other hand, Algorithm 1 in this paper randomly selects a nonterminal node for perturbation. Thus, in some cases, the algorithm may operate in a breadth-first manner, and we will not gain much from the inference rules.

Therefore, we redesign Algorithm 1 so that it operates in a modified *depth-first* manner, called *hybrid* in Section V. Before going to Section V, we study the concept of *projection machines* in Section IV. The projection idea allows us to compute initial UIO sequences for some machines in polynomial time. Thus, if the projection idea yields all UIO sequences, there is no need to even partially construct a UIO tree. However, if the projection machines are found to be inadequate in deciding about the existence of UIO sequences, then we finally resort to a partial construction of UIO trees in Section V.

IV. Projection

Informally, a projection graph is a subgraph of a given graph such that all edges have identical labels. Thus, we obtain a family of projections corresponding to the set of distinct edge labels in a machine. From the point of input and output, any two projection graphs are different. This difference allows us to quickly extract UIO sequences in polynomial time from projection graphs without any comparison involving other projection graphs. In the following, we formalize the projection concept.

Definition: Given a graph $G = (V, E)$ and a label $\gamma = a/b$ of input/output pair, a *projection graph* $G(\gamma) = (V', E')$ is the subgraph of G, such that:

$$V' = \{\text{head}(e), \text{tail}(e) | \gamma = \text{label}(e) \wedge e \in E\}$$

and

$$E' = \{e | \gamma = \text{label}(e) \wedge e \in E\}.$$

Since the same pair of input/output is associated with all of the edges in $G(\gamma)$, we refer to the input and output symbols in $G(\gamma)$ as $\text{input}[G(\gamma)] = \text{input}(\gamma)$ and $\text{output}[G(\gamma)] = \text{output}(\gamma)$, respectively. A projection graph is an edge-induced graph with all the edges having the same input/output label.

Definition: Given a graph $G = (V, E)$ and the set of distinct labels $L = \{\gamma | \gamma = \text{label}(e) \wedge e \in E\}$, a *projection family* is defined as the set of projection graphs: $\mathbf{G}(L) = \{G(\gamma) | \gamma \in L\}$.

A projection graph is a *forest* of components. For a given machine G, let $G(\gamma) = (V', E')$ be a projection with respect to the input/output pair γ. Then, we denote the forest for a projection as $G(\gamma) = \{G_1(\gamma), \cdots, G_m(\gamma)\}$.

In the following, we define two types of components. Given a projection forest $G(\gamma) = \{G_1(\gamma), \cdots, G_m(\gamma)\}$, $\text{source}[G_i(\gamma)] = \{v | \text{in}(v) = 0 \wedge v \in V'\}$. Similarly, $\text{sink}[G_i(\gamma)] = \{v | \text{out}(v) = 0 \wedge v \in V'\}$.

Definition: $G_i(\gamma) \in G(\gamma)$ is said to be *linear* if $|\text{source}[G_i(\gamma)]| \geq 1$ and $|\text{sink}[G_i(\gamma)]| = 1$.

Definition: $G_i(\gamma) \in G(\gamma)$ is said to be *cyclic* if $|\text{source}[G_i(\gamma)]| \geq 0$ and $|\text{sink}[G_i(\gamma)]| = 0$.

Remark 4: Since the original machine is deterministic, no vertex of a projection has out-degree more than one. Thus, no cyclic component has nested cycles.

Definition: A *maximal path* or simply *path* starting with v and denoted by P_v is defined as the sequence of edges which terminates either at a sink vertex or at another vertex already appearing on the path.

A path is said to be linear if it terminates at a sink vertex. Otherwise, it is cyclic. The number of edges in a path denotes the length of the path and is denoted by $\text{length}(P_v)$. We associate an input sequence $I(P_v)$ with the start vertex of the path. $I(P_v)$ is the input sequence that takes the projection component from the first state to the last state of the path.

In the following, we state some properties that allow us to readily identify some states having or not having UIO sequences.

Property 1: If a projection contains exactly one transition, then the head state of the transition has a UIO sequence.

Proof: Trivial. □

Property 2: In a projection $G(\gamma) = (V', E')$, if there is exactly one vertex v such that P_v is linear and $I(P_v)$ is the longest among all linear paths, then some states in V' possess UIO sequences with an input sequence of length $(L_1 + L_2)$, where $L_1 = \text{length}(P_v)$, and

$L_2 = 0$ if all paths in the component are linear;

$L_2 = 1$ if there is a cyclic path.

Proof: We prove the lemma by showing that UIO (v) is given by $x/\lambda(v, x)$, where x is a string of input consisting of $(L_1 + L_2)$ $\text{input}(\gamma)$ symbols. We consider two cases. In the first case, projection $G(\gamma)$ contains only linear paths and in the second case, $G(\gamma)$ contains both linear paths and cyclic paths.

Case 1: In this case $L_1 = \text{length}(P_v)$ and $L_2 = 0$. If we apply x to state v, we get an output sequence $\lambda(v, x)$ consisting of L_1 $\text{output}(\gamma)$ symbols. When we apply x to a state $v' \in V' - \{v\}$, we get a different output string because $I(P_{v'})$ is a proper subsequence of $I(P_v)$. The two strings $\lambda(v, x)$ and $\lambda(v', x)$ will have their kth symbol different, where $k = \text{length}[I(P_v)] - \text{length}[I(P_v')] + 1$. This is because the kth input will enable a transition in $G(\gamma')$, where $\text{output}(\gamma') \neq \text{output}(\gamma)$.

Case 2: In this case, $L_1 = \text{length}(P_v)$ and $L_2 = 1$. Let $G_c = (V'', E'')$ be a cyclic component in the given projection. For every pair of states $v_i', v_j' \in V''$, $\lambda(v_i', x) = \lambda(v_i', x)$. However, $\lambda(v, x)$ differs from $\lambda(v_i', x)$ in the last output because the last input in sequence x enables a transition in a different projection $G(\gamma')$, where $\text{output}(\gamma') \neq \text{output}(\gamma)$.□

Property 3: If a projection consists of only linear components, then every vertex v, such that $\text{length}(P_v)$ is unique among all v in the components, has a UIO sequence.

Proof: The proof is similar to *Case 1* in Property 2. □

Property 4: If s_i is a converging state with respect to each input in I, then s_i does not have any UIO sequence.

Proof: Trivial. □

Example 6: Machine $G2$ and its projections are shown in Fig. 3. The set of edge labels is given by $L = \{1/1, 0/0, 1/0, 0/1\}$. The four projections are $G2(1/1)$, $G2(0/1)$, $G2(1/0)$, and $G2(0/1)$. Now, we analyze these projections one by one.

Referring to $G2(1/1)$, we have $\text{length}[I(P_B)] = 3$, $\text{length}[I(P_A)] = 2$, $\text{length}[I(P_D)] = 1$. Thus, each of states $\{B, A, D\}$ has a UIO sequence of length 3. That is UIO$(B) = 111/\lambda(B, 111)$, UIO$(A) = 111/\lambda(A, 111)$, and UIO$(D) = 111/\lambda(D, 111)$. Referring to $G2(0/0)$, we have $\text{length}[I(P_D)] = 2$, $\text{length}[I(P_B)] = 1$, and $\text{length}[I(P_A)] = 1$. Since both states B and A have identical path lengths, no UIO sequence for these two states can begin with input/output 0/0. However, UIO$(D) = 00/\lambda(D, 00)$. Referring to $G2(1/0)$, UIO$(C) = 1/0$, and referring to $G2(0/1)$, state C has another UIO sequence given by UIO$(C) = 0/1$. (**End of Example**)

So far, we have studied some properties of the projection machines to readily identify which states have and which states do not have UIO sequences. Using the inference rules, we can obtain UIO sequences for some other states. However, there are machines for which the projection technique does not yield any UIO sequences. In addition, projections and

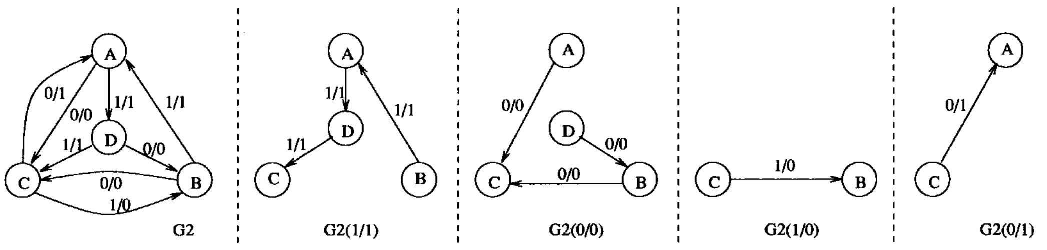

Fig. 3. Machine $G2$ and its projections.

inference rules may not yield UIO sequences for some states. In that case, we do not know whether or not those states have UIO sequences. Thus, we must resort to the direct method of UIO computation to determine the presence or absence of UIO sequences for those states. However, if we do not construct a UIO tree selectively, saving in time due to the application of inference rules may not be significant. In the following section, we discuss how to construct a UIO tree efficiently.

V. Efficient Construction of a UIO Tree

This section is organized in four parts. In the first part, we give the general structure of constructing a UIO tree in three ways: *breadth-first, depth-first,* and *hybrid*. In the second part, we incorporate the ideas of projection and inference into the hybrid approach, and present an efficient UIO-construction algorithm. In the third part, we define the idea of an *essential subtree* that allows us to compare the average-case performance of the algorithm with its worst case. In the last part, we give a detailed example.

A. Three Approaches to UIO Tree Construction

1) Breadth-First Approach: In the breadth-first approach, we compute all input/output sequences of length 1, of length i, of length $(i+1)$, and so on, until a sequence becomes a UIO sequence or we reach a node that has already been generated previously. In the following, we give an outline of the breadth-first approach.

We manage three buckets: B_T, B_i, and B_{i+1}. Bucket B_T contains all terminal nodes. A node need not be perturbed if it is a singleton node, or if it is a node that already appears before on the path from the initial node to the node under consideration, or if UIO sequences for all states have been computed. Bucket B_i contains nodes ψ such that the path from the initial node to ψ is of length i. Bucket B_{i+1} contains nodes ψ such that the path length is $i+1$. The perturbation process is as follows.

Step 1: Initialize $i = 0$, B_i with the initial node, and $B_{i+1} = \phi$.

Step 2: If B_i is empty, then stop; else execute the following for each member $\psi \in B_i$.

If ψ is a terminal node, then $B_T = B_T \cup \{\psi\}$; else $B_{i+1} = B_{i+1} \cup \{\text{pert}(\psi, L)\}$;

Update the UIO tree with the set of new nodes $\{\text{pert}(\psi, L)\}$, and let $B_i = B_i - \{\psi\}$

Step 3: Increment i, set $B_{i+1} = \phi$, and go to Step 2. □

2) Depth-First Approach: In the depth-first approach, a path is expanded until we reach a terminal node. Next, we choose another path to expand until a terminal node is reached.

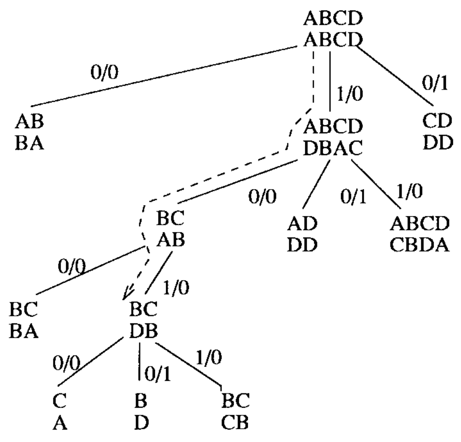

Fig. 4. An example of hybrid approach.

However, selection of a node to perturb in order to expand a path is not arbitrary. Rather, we select a node for perturbation by backtracking from a terminal node. The backtracking is similar to the way logic programs, such as Prolog, construct a search space. The depth-first approach is outlined below.

Step 1: Initialize the tree with the initial node. Let ψ be a selected node.

Step 2: Perturb ψ along a path until a terminal node is reached.

Step 3: Backtrack from the terminal node reached in Step 2 until an incompletely perturbed node ψ. If no such node is found or if the UIO sequences for all states have been computed, then stop; else go to Step 2. □

In the depth-first approach, there is a likelihood that UIO sequences of longer length are computed before shorter UIO sequences are computed for the same states. Readers can verify this by referring to the UIO tree of Fig. 1. This is largely avoided in the hybrid approach explained below.

3) Hybrid Approach: The hybrid approach is a combination of a depth-first and breadth-first approach. It is largely a depth-first approach. However, while moving along a path, we perturb each new node for all edge labels, and select one of them for further perturbation. The hybrid approach is outlined below.

Step 1: Initialize the tree with the initial node. Select node ψ for perturbation.

Step 2: Compute $\text{pert}(\psi, L)$.

Step 3: Arbitrarily choose a nonterminal $\psi' \in \text{pert}(\psi, L)$. If no such node is found, then go to Step 4, else set $\psi = \psi'$ and go to Step 2.

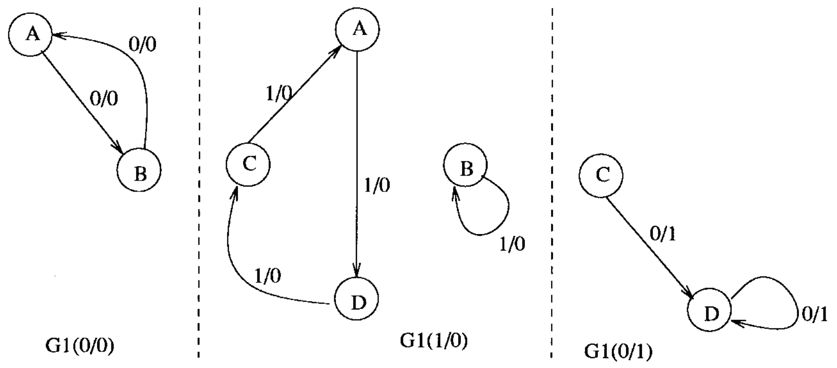

Fig. 5. Projections of machine $G1$.

Step 4: Backtrack from ψ until a node ψ' is reached such that all members of pert(ψ', L) have not been perturbed. If no such node is found, then stop, else select a member $\psi \in$ pert(ψ', L) and go to Step 2. □

In this approach, shorter UIO sequences are detected earlier than longer sequences for the same state. An example of expanding the UIO tree for machine $G1$ using the hybrid technique is shown in Fig. 4. The dotted line shows an arbitrary path constructed in a depth-first manner. Other nodes not on the dotted path are due to the hybrid nature of expansion. In the depth-first approach, the algorithm perturbs $(B/C, C/B)$ before completely perturbing its predecessor $(B/D, C/B)$. Thus, the depth-first approach will generate a longer UIO sequences for states B and D before generating the shorter ones. However, in the hybrid approach, since we completely perturb all nodes along the chosen path, shorter UIO sequences will be detected before longer ones.

B. Reduction of the Search Space

In this subsection, we augment Algorithm 1 with the ideas of projection, inference, and hybrid manner of constructing a UIO tree to obtain Algorithm 2. An informal description of Algorithm 2 is as follows. Initially, nothing is known about the existence of UIO sequences. Thus, in the first step, we use the projection technique to possibly compute a UIO sequence for some states, and decide if some states do not have UIO sequences. In case the projections yield some UIO sequences, we use the inference rules to compute UIO sequences for some more states. For each state, if we know whether or not the state has a UIO sequence, then there is no need to construct a UIO tree. However, if we have some states for which no such decision can be taken, then we construct a UIO tree. Construction of the UIO tree is guided by the concept of *rule base* and *hybrid* manner of expanding the tree.

Let $\mathrm{no}(S) \subseteq S$ denote a set of states known to have no UIO sequence. Set $\mathrm{no}(S)$ can be computed using Property 4 of projection machines. Let possible$(S) = S - \mathrm{no}(S)$ denote the set of states possibly having UIO sequences. Now we formulate Algorithm 2 as follows.

Algorithm 2:

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

Output: Reduced UIO tree and UIO sequences for all possible states.

Procedure: Execute the following steps.

Step 1: Compute $\mathbf{G}(L)$.

Step 2: If possible, compute UIO sequences for some states, and apply the inference rules. If UIO sequences for all states in possible(S) are constructed, then stop.

Step 3: Let Ψ be the set of nodes denoting path vectors. Initially, Ψ contains the initial node marked as *nonterminal*. Select a node $\psi \in \Psi$.

Step 4: Compute pert(ψ, L) and update the tree. If pert(ψ, L) contains singleton vectors, then apply the inference rules to compute additional UIO sequences. If a UIO sequence is found for each state in possible(S), then stop.

Step 5: Select any nonterminal node $\psi' \in$ pert(ψ, L) for perturbation. If no such node is found, then go to Step 6, else set $\psi = \psi'$ and go to Step 4.

Step 6: Backtrack from ψ until a node ψ', such that all members of pert(ψ', L) have not been perturbed. If no such node is found, then stop; else select a member $\psi \in$ pert(ψ', L) and go to Step 4.

End of Algorithm 2

The definition of a nonterminal node follows from pruning conditions C1 and C2 in Section II.

C. Performance Measure

Let us analyze the worst case scenario. If some states in possible(S) do not have UIO sequences, then the algorithm will construct the complete UIO tree. However, if all states in possible(S) have UIO sequences, then Algorithm 2 constructs a very small subtree. Algorithm 2 randomly computes some initial UIO sequences. Therefore, it is meaningful to analyze the average case performance of the algorithm. Since the performance of the algorithm is dependent on machine structure, obtaining an expression for the average behavior in terms of state and transitions may be a difficult task.

On the other hand, it is useful to study the minimum and maximum computation that the algorithm must do in case a machine has UIO sequences for all states. The idea of an *essential subtree*, defined below, is used to study the average performance of the algorithm.

Definition: An *essential subtree* of a complete UIO tree is a subtree, rooted at the initial node, that contains UIO sequences for enough states, so that UIO sequences for other states can be inferred, if they exist.

Because of the randomness in the algorithm, it is possible that a UIO tree may contain several essential subtrees of different sizes. Thus, in order to gain knowledge about the average performance of the algorithm, we need to compare the minimum and maximum size of essential subtrees with the complete UIO tree. Here, *size* denotes the number of nodes in a UIO tree or in a subtree. In Section VII-B, we compare these sizes for a number of different machines.

D. Example of Using Algorithm 2

We apply Algorithm 2 on machine $G1$. The set of distinct edge labels for $G1$ is given by $L = \{0/0, 1/0, 0/1\}$, and the three projections $G1(0/0)$, $G(1/0)$, and $G(0/1)$ are shown in Fig. 5. Now, we compute the unique predecessors for all states as follows:

$$\begin{aligned} S^u(A) &= \{B, C\} \\ S^u(B) &= \{A\} \\ S^u(C) &= \{D\} \\ S^u(D) &= \{A\}. \end{aligned}$$

Therefore, the rule base is given by:

$$\begin{aligned} \text{UIO}\,(B) &= \text{IO}\,(B, A)@\text{UIO}\,(A) \\ &= 0/0@\text{UIO}\,(A) \\ \text{UIO}\,(C) &= \text{IO}\,(C, A)@\text{UIO}\,(A) \\ &= 1/0@\text{UIO}\,(A) \\ \text{UIO}\,(A) &= \text{IO}\,(A, B)@\text{UIO}\,(B) \\ &= 0/0@\text{UIO}\,(B) \\ \text{UIO}\,(D) &= \text{IO}\,(D, C)@\text{UIO}\,(C) \\ &= 1/0@\text{UIO}\,(C) \\ \text{UIO}\,(A) &= \text{IO}\,(A, D)@\text{UIO}\,(D) \\ &= 1/0@\text{UIO}\,(D). \end{aligned}$$

The inference graph for machine $G1$, shown in Fig. 2, has a cycle containing all of the states. Thus, if we can compute a UIO sequence for one state, we can infer all other UIO's.

From the projection machines, we cannot determine a subset of states having or not having UIO sequences. Thus, $\text{no}(S) = \phi$ and $\text{possible}(S) = \{A, B, C, D\}$. Now, we need to construct a UIO tree starting with the initial node $\psi_1 = (A/A, B/B, C/C, D/D)$. Perturbing the initial node by the three elements of L, we obtain three nodes: $(A/B, B/A)$, $(C/D, D/D)$, $(A/D, B/B, C/A, D/C)$. Node $(C/D, D/D)$ is a terminal node because it has a homogeneous tail vector. Therefore, we select either of the other two nodes for perturbation. Let us select node $(A/B, B/A)$. We can perturb node $(A/B, B/A)$ using edge labels 0/0 and 1/0 to obtain nodes $(A/A, B/B)$ and $(A/B, B/D)$, respectively. Node $(A/A, B/B) \subseteq \psi_1$. Thus, we perturb node $(A/B, B/D)$ using edge labels 0/0, 0/1, and 1/0 to obtain (A/A), (B/D), and $(A/B, B/C)$, respectively. Since nodes (A/A) and (B/D) indicate the presence of UIO sequences for states A and B, respectively, the algorithm applies the inference rules to compute additional UIO sequences. Since we know the UIO sequences for A and B, we can apply rule $\text{UIO}\,(C) = \text{IO}\,(C, A)@\text{UIO}\,(A)$ to compute a UIO sequence for state C. Also, using rule $\text{UIO}\,(D) = \text{IO}\,(D, C)@\text{UIO}\,(C)$, we can compute a UIO sequence for state D. Table IV summarizes the four UIO sequences. The reduced UIO tree is shown in Fig. 6. Incidentally, Fig. 6 shows the minimum essential subtree for $G1$. The maximum essential subtree for G1 is shown in Fig. 7. The complete UIO tree contains 34 nodes, whereas the minimum and maximum essential subtrees contain 9 and 15 nodes, respectively.

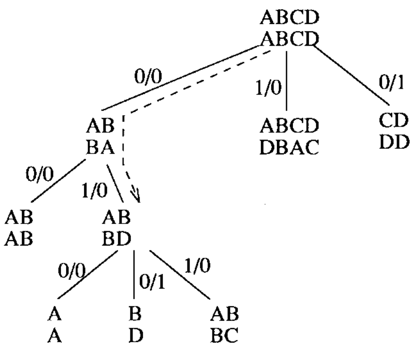

Fig. 6. Minimum essential subtree for machine $G1$.

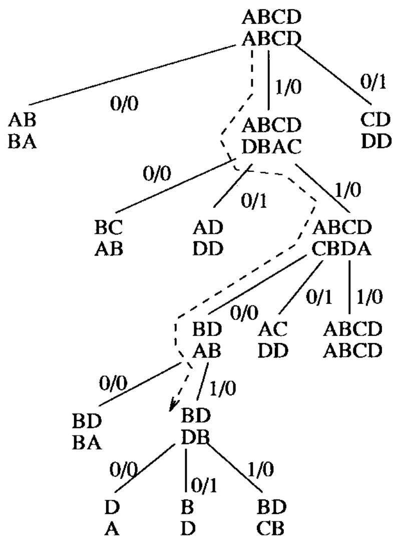

Fig. 7. Maximum essential subtree for machine $G1$.

TABLE IV
SOME UIO SEQUENCES FOR MACHINE $G1$

A	010/000
B	010/001
C	1010/0000
D	11010/00000

VI. CONVERGING FSM AND UIO SEQUENCES

Converging states were first observed by Miller and Paul [10]. If there are states $s_1, \cdots, s_j$ going into some state s_k with the same input/output label a/b, states $s_1, \cdots, s_j$ are called *converging states* and state s_k is called the *convergent state*. The transitions $(s_1, s_k; a/b), \cdots, (s_j, s_k; a/b)$ are called *converging transitions*. A machine M is said to be *nonconverging* if there are no two transitions entering the same state with the same input/output label. With this understanding of converging transitions and FSM's, we study the existence of UIO sequences in a class of machines. A terminal node of a UIO tree is one of the following three types:

1) *singleton node* satisfying condition C1;
2) *nonsingleton homogeneous node* satisfying condition C1;
3) *repetitive node* satisfying condition C2.

Lemma 1: If $G = (V, E)$ is a CDRS machine, there is at least one singleton node in the UIO tree of G.

Proof: We prove the lemma by contradiction. Assume that there is no singleton node in the UIO tree of G. Thus, a terminal node in the UIO tree is of either *repetitive* type or *nonsingleton homogeneous* type. In the following, we prove the lemma for FMS's with $2, 3, \cdots$ states.

a) *Two-State FSM:* Let $S = \{A, B\}$ be the set of states in G. Therefore, all of the terminal nodes of the UIO tree are of the form $(A/X, B/Y)$, where X and Y are instantiated with the elements of S. Since the state pair (A, B) appears in all IV components of the terminal node, we have $\lambda(A, I') = \lambda(B, I'), \forall I' \in I^*$. Thus, we conclude that A and B are *equivalent states*. However, this conclusion contradicts our initial assumption that the given machine is reduced. Since G is said to be reduced, the other assumption that the UIO tree of G does not contain any singleton node must be false.

b) *Three-State FSM:* Let $S = \{A, B, C\}$ be the set of states in G. Therefore, all of the terminal nodes of the UIO tree are of the form $(A/X_1, B/Y_1, C/Z_1)$, $(A/X_2, B/Y_2)$, $(A/X_3, C/Y_3)$, and $(B/X_4, C/Y_4)$, where X_i, Y_i, and Z_i are instantiated with the elements of S. All possible perturbations of a node with three elements, such as $(A/X_1, B/Y_1, C/Z_1)$, generate a set of nodes with two members and three members. Let us perturb node $(A/X_1, B/Y_1, C/Z_1)$ using edge labels a_i/b_j and a'_i/b'_j to obtain nodes $(A/X_2, B/Y_2)$ and $(A/X_3, C/Y_3)$, respectively. Now, we have the following two cases.

Case 1—$a_i = a'_i$: Since we obtain two nodes $(A/X_2, B/Y_2)$ and $(A/X_3, C/Y_3)$, obviously, $b_j \neq b'_j$. Therefore, we can argue that there exists an input sequence I' such that $\lambda(A, I') \neq \lambda(B, I')$, which means that for some input sequence I', states B and C are separable from each other. However, the presence of state A in both $(A/X_2, B/Y_2)$ and $(A/X_3, C/Y_3)$ suggests that machine G is *nondeterministic*. Since G is a deterministic machine, the perturbation process cannot obtain two nodes $(A/X_2, B/Y_2)$ and $(A/X_3, C/Y_3)$ from $(A/X_1, B/Y_1, C/Z_1)$ using two edge labels whose input components are identical.

Case 2—$a_i \neq a'_i$: In this case, the equality or the lack of it between b_j and b'_j does not matter. Using the *two-state FSM* case, we can argue that states A and C are equivalent states. However, this contradicts our assumption that G is a reduced machine.

The above two cases suggest that the assumption that the UIO tree of a nonconverging CDRS machine does not contain singleton nodes leads to contradictions. Therefore, the assumption that the UIO tree of G does not contain any singleton node must be false.

c) *Any-State FSM:* Using the arguments provided for two-state and three-state FSM's above, one can reach similar contradictions for FSM's with $4, 5, \cdots$ states.

Therefore, the UIO tree of a CDRS machine must contain at least one singleton node. □

Theorem 2: If $G = (V, E)$ is a nonconverging CDRS machine, each state of G has UIO and multiple UIO sequences, where the degree of multiplicity is equal to the number of inputs to G.

Proof: Lemmas 1 ensures that the UIO tree of G has at least one singleton node. That is, at least one state v_i has a UIO sequence. Since G is nonconverging, the inference graph of G is G itself, which is also strongly connected. Therefore, using the inference mechanism, we can obtain UIO sequences for all other states from the UIO sequence of v_i. Additionally, for each outgoing edge $(v_j, v_k; a/b)$ from state v_j, we can derive a UIO sequence for v_j from that of v_k. □

From Theorem 2, it is evident that nonconvergence of a CDRS machine is a *sufficient* condition for the existence of UIO sequences for each state of the machine. However, nonconvergence is not a necessary condition. For example, although machine $G1$ contains converging transitions, each state of the machine has a UIO sequence. In spite of the presence of UIO sequences for all states in some converging FSM's, we believe that as an FSM contains more and more converging transitions, the chances of all states having UIO sequences become fewer and fewer. If an FSM contains an "overwhelming" number of convergent states, it is very likely that some states do not have UIO sequences. Therefore, it is desirable to find the cutoff point on converging transitions for the existence of UIO sequences for all states.

Theorem 3: If the inference graph of a converging CDRS machine G is strongly connected, every state of G has a UIO sequence.

Proof: According to Lemma 1, the UIO tree of the given CDRS machine has at least one singleton node. Thus, at least one state of the machine has a UIO sequence. Since the inference graph is strongly connected, one can infer UIO sequences for all other states. □

Remark 3: All of the states of a converging CDRS machine do not have multiple UIO sequences. This follows from the fact that a converging transition cannot be used in an inference rule.

Theorem 4: Let $G = (V, E)$ be a convergent CDRS machine and let $\text{IG}(G) = (V, E')$, where $E' \subset E$, be its inference graph. If a state v_i does not have any outgoing edges in $\text{IG}(G)$, then v_i does not have any UIO sequence.

Proof: An inference graph $\text{IG}(G)$ is obtained from G by simply deleting the converging edges from G. Thus, if v_i has no outgoing edges in $\text{IG}(G)$, all of the outgoing edges from v_i in G were converging edges. Consider an edge $(v_i, v_j; a/b)$. Even if state v_j has a UIO sequence, state v_i cannot have a UIO sequence starting with edge label a/b because there is another state v_k and an edge $(v_k, v_j; a/b)$ which do not allow

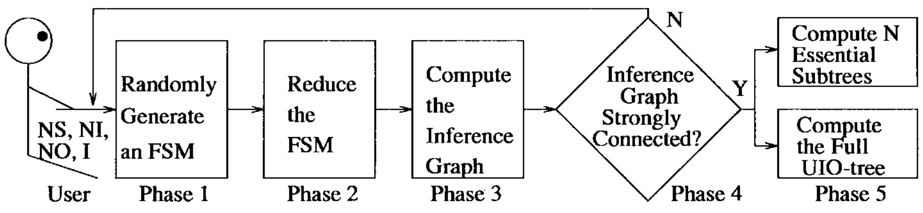
NS, NI, NO, I
User
Randomly Generate an FSM
Phase 1
Reduce the FSM
Phase 2
Compute the Inference Graph
Phase 3
Inference Graph Strongly Connected?
N
Y
Phase 4
Compute N Essential Subtrees
Compute the Full UIO-tree
Phase 5

Fig. 8. Overview of the tool.

us to apply the inference rule to compute a UIO sequence for v_i using UIO (v_j). Applying this argument to each converging edge outgoing from v_i, we conclude that there is no UIO sequence for v_i. □

Before going to the next subsection, we summarize the observations made in this subsection on a CDRS machine G as follows.

- If G is nonconverging, then each state of G has UIO and multiple UIO sequences.
- If G is converging and its inference graph is strongly connected, then each state of G has at least one UIO sequence.
- If G is converging and some states in its inference graph do not have outgoing edges, then those states do not have UIO sequences.

VII. Performance of the Algorithm

A. Development of a Tool

An overview of our tool is presented in Fig. 8. The tool is implemented in *five* phases. In Phase 1, we generate a random FSM. The user supplies three input parameters to the FSM generator: the number of states, the number of inputs, and the number of outputs. The FSM is represented as a set of transitions, where each transition is a tuple of the form: *trans(From, To, Input, Output)*. The pseudocode of this phase is given below. We implemented the FSM generator in C.

```
/* Pseudocode to generate a random FSM. */
Inputs to the random FSM generator: NS, NI, NO.
NS is the number of states in the FSM.
NI is the number of inputs to the FSM.
NO is the number of outputs of the FSM.
An FSM is represented by a set of transitions
of the form: trans(from, to, input, output).

for from = 1 to NS do [
    for input= 1 to NI do [
        to     = random number in the range (1, NS)
        output = random number in the range (1, NO)
        write trans(from, to, input, output)
    ]
]
/* End of pseudocode to generate a random FSM. */
```

We implemented Phases 2–5 in Prolog. These four phases of the tool consist of about 220 rules. In Phase 2 we reduce the FSM generated in Phase 1. To reduce an FSM, we implemented the algorithm of Hopcroft [1], [2]. In Phase 3, we obtain the inference graph from the reduced FSM. In Phase 4, we check whether the inference graph is strongly connected. In this phase, we implemented the algorithm in [2] for computing the number of strongly connected components in a directed graph. We considered only strongly connected inference graphs to compare the size of essential subtrees with the full size of a UIO tree. In Phase 5, we compute the full UIO tree and the essential subtrees from a given FSM.

TABLE V
Comparing the Sizes of Essential Subtrees With the Full UIO-Trees

FSM	$\|S\|$	$\|I\|$	$\|O\|$	$\|E\|$	No. of of ES	Max. size of ES	Size of full UIO-tree
1 [3]	4	2	2	8	6	15	34
2 [3]	4	2	2	8	24	5	21
3 [7]	4	2	2	8	6	13	37
4 [11]	6	2	2	12	24	11	84
5 [12]	6	2	2	12	6	41	856
6	20	5	5	100	100	48	8170
7	30	5	5	150	100	44	20,000+
8	15	25	15	375	50	242	10,000+
9	100	10	5	1000	100	101	10,000+
10	50	20	10	1000	50	201	10,000+

E = set of transitions in an FSM, ES = Essential Subtree

B. Experimental Result

Here, we compare the maximum size of an essential subtree with the full size of the UIO tree for a number of FSM's. We consider both small and large FSM's in our experiment. At first, we computed the UIO trees from the small FSM's by hand. The small UIO trees were very useful in debugging our tool. Large FSM's with hundreds of transitions were generated in Phase 1 of our tool. We computed all of the essential subtrees for the small FSM's. However, computing all of the essential subtrees for a large FSM with hundreds of transitions is computationally prohibitive. Therefore, we computed about 50–100 random essential subtrees for each large FSM. Our experimental results are summarized in Table V. Small machines 1–5 were picked up from published works. Machines 6–10 were generated using our tool. The sizes of essential subtrees are very small compared to the sizes of the full UIO trees. This demonstrates the superiority of our new approach to UIO computation over the direct approach. Thus, our approach to UIO computation will be very useful in practice, where one may have to deal with machines with hundreds of transitions.

VIII. Impact on Test Sequence Length

Since the ultimate use of UIO sequences lies in the generation of test sequences, we study how UIO's generated using our technique affect the length of test sequences. Longer test sequences incur additional cost. Depending on the interpretation of the cost of executing a transition, the penalty for longer test sequences may not be acceptable. For instance, if the execution of a state transition leads to a physical activity, say in a process control system, etc., the monetary loss due to repeated execution of the physical process may be too much. If the test sequences are very long due to longer UIO sequences, our method will not be attractive to the practitioners.

It is obvious that when we use the inference mechanism, the derived UIO's get longer and longer. For example, referring to Fig. 1(a), if the algorithm first finds UIO(B), the inference mechanism will compute UIO(A), UIO(C), and UIO(D) in that order. Hence, if the length of UIO(B) is L, then the lengths of UIO(A), UIO(C), and UIO(D) are $L+1$, $L+2$, and $L+3$, respectively. If the inference mechanism is used in a straightforward manner, in the worst case, we may obtain UIO(s_i) of length $|\text{UIO}(s_j)|+N-1$, where N is the number of nodes in the graph, and UIO(s_j) is the first sequence the algorithm computes. The increase in the lengths of UIO sequences by order $O(N)$ is highly undesirable. In this section, we propose an efficient way of using the inference mechanism such that the increase in the length of UIO sequences is limited to $O(d_{G'})$, where G' is an inference graph and $d_{G'}$ is the *diameter*[2] of G'. On an average, $d_{G'} \ll N$. Therefore, the impact of the inference mechanism on the length of UIO sequences is significantly reduced.

In this section, we informally explain how the UIO generation technique can be fine tuned for two test generation techniques [8], [10], so that the UIO's do not lead to a significant increase in the length of test sequences. The basic ideas in fine tuning the algorithm are to selectively generate UIO's depending on the need and to trim a chain of inference rules.

A. For the Aho–Dahbura–Lee–Uyar Method

We first explain the structure of a test sequence generated by Aho, Dahbura, Lee, and Uyar [8]. Their test sequence is of the form

$$(ri) \cdots [\text{TEST}(v_{i_1}, v_{i_1}; L_1)] \cdots \\ [\text{TEST}(v_{i_{|E|}}, v_{j_{|E|}}; L_{|E|})] \tag{1}$$

where ri is a reset input, E is the set of transitions, and TEST$(v_i, v_j; a_k/o_\ell)$ is an input sequence to test transition $(v_i, v_j; a_k/o_\ell)$. TEST$(v_i, v_j; a_k/o_\ell)$ is given by a_k@UIO(v_j). They optimize the length of expression (1) by using *Chinese rural postman tours*, such that the minimum cost sequence contains the subsequence TEST$(v_i, v_j; L_k)$ for each transition $(v_i, v_j; L_k) \in E$ and no two such subsequences are overlapped. The optimization method requires the test designer to explicitly compute all TEST segments, and therefore UIO's, for *all* states.

If we blindly derive UIO's for all other states from the UIO of one state using a sequence of inference rules, we may generate UIO's with extra length $O(N)$. To reduce the extra

[2]The diameter of a graph is the maximum length of the shortest paths between all pairs of nodes.

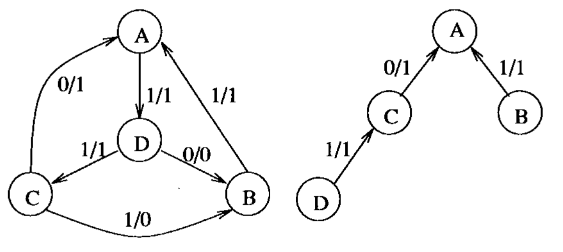

Fig. 9. (a) Inference graph of $G2$. (b) Breadth-first unfolding of the inference graph.

length, we propose a method to judiciously use the inference rules using the following five steps.

J1) Given an inference graph G', reverse the edge directions with no change to their input/output labels.

J2) Obtain a tree, say T, rooted at v [v is the state for which we have found UIO(v)] by unfolding G' in a breadth-first manner [2].

J3) Reverse the directions of the edges of the tree obtained in step J2.

J4) For each edge in T, derive an inference rule similar to the one explained in Section III.

J5) Using the rule base obtain in step J4, derive UIO's for all possible states starting with the seed UIO(v).

The main idea in the above procedure is to trim the length of the chain of inference rules. The breadth-first unfolding leads to a balanced approach to using the inference rules. It can be shown that the height of T is equal to the diameter of G'. Thus, this approach to using the inference rules limits the extra lengths of UIO sequences to $d_{G'}$. The novelty of this approach is that we exploit the structure of an inference graph to reduce the length of the inferred UIO's.

Fig. 9(a) shows the inference graph of machine $G2$, and Fig. 9(b) shows an unfolding of the inference graph using the breadth-first approach. The height of the tree in Fig. 9(b) is equal to 2, which is the diameter of the inference graph.

Each UIO sequence may increase by an amount of $d_{G'}$. Since a UIO sequence is explicitly used in a test sequence of the form (1), the increase in length of a test sequence is of the order $O(|E| \cdot d_{G'})$, where E is the set of transitions in the specification FSM.

B. For the Miller–Paul Method

The Miller–Paul method [10] derives test sequences based on whether or not the given FSM contains converging transitions. On the one hand, if the FSM does not contain converging transitions, the test generation method needs a UIO for only *one* state. On the other hand, if the FSM contains converging transitions, the method needs UIO$_{i,j}$ *for all* $v_i, v_j \in V$, where UIO$_{i,j}$ is the minimum length UIO sequence from v_i to v_j. The minimality requirement is for reducing the length of the test sequence.

Thus, depending on the structure of the given FSM, a test designer needs to compute UIO sequences for either one or all states. If we need UIO sequences for all states, those can be computed as explained in Section VIII-A using an unfolded inference graph. However, if we need a UIO sequence for

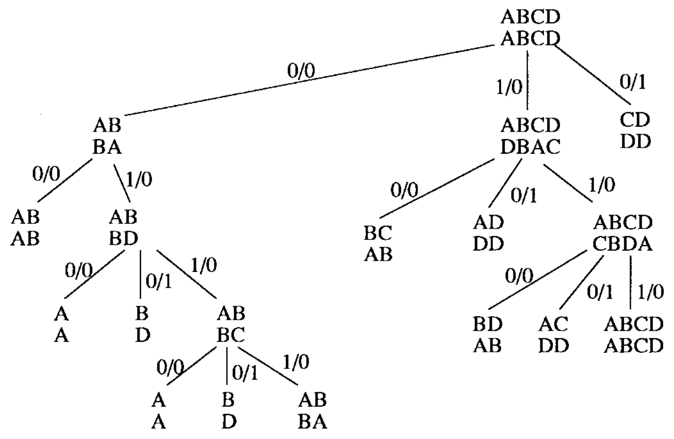

Fig. 10. Goal-directed UIO tree for $G1$.

only one state, say v_i, then the idea of Section VIII-A will not be very useful in reducing the length of UIO(v_i). In the following we discuss another fine-tuned method of computing UIO(v_i) *without using* the inference mechanism. Assuming that our goal is to compute UIO(v_i), rather than an arbitrary UIO(v_j), we explain how to efficiently construct a UIO tree so that we obtain UIO(v_i) from the UIO tree. Obviously, if our first UIO sequence from the UIO tree is the desired UIO(v_i), there is no further need to use the inference rules. Hence, we avoid the disadvantage of using the inference mechanism. In the following we informally explain how the desired UIO(v_i) can be computed.

The main ideas are the following. First, add the following new pruning condition C3 to C1 and C2 of Section II: C3: $v_i \notin IV(PV')$.

Second, replace expression (C1 or C2) in Step 4 of Algorithm 1 by (C1 or C2 or C3). The resulting effect is that we perturb only those path vectors which have the potential of producing UIO(v_i), if it exists. Trivially, if $v_i \notin$ IV(PV$'$), then perturbation of PV$'$ will not give us UIO(v_i). We call the new UIO tree a *goal-directed* UIO tree because our goal is to derive UIO(v_i). The size of the UIO tree can further be reduced by using the depth-first approach.

As an example, we show the goal directed UIO tree for FSM $G1$ in Fig. 10 to compute UIO(A). In spite of constructing a goal-directed essential subtree in the depth-first approach, its average size will be larger than an essential subtree obtained without pruning condition C3. However, the main point here is that we do not use the inference mechanism to obtain a UIO sequence for the desired state.

Now, we analyze the impact of the inference mechanism on the length of a test sequence designed using the Miller–Paul method.

- If the specification FSM does not contain converging transitions, then the inference mechanism has no impact on the test sequence length. The extra length of a test sequence is the difference between the minimal length UIO sequence for the desired state and the length of the arbitrarily chosen UIO sequence for the same state.
- If the specification FSM contains converging transitions, the effect of the inference mechanism is similar to the case studied for the Aho–Dahbura–Lee–Uyar method.

Remark 6: Although the impact of the inference mechanism on test sequence length is of order $O(|E| \cdot d_{G'})$, it is likely to be much less on the average. This is because of the following two reasons.

- The difference between the lengths of two UIO sequences computed for the same state using and without using the inference mechanism may not be as large as $d_{G'}$.
- Excluding the first application of an inference rule in the UIO derivation process, each subsequent application leads to one test segment subsuming another test segment. Such overlapping of test segments can be exploited in reducing the total length of a test sequence.

IX. Concluding Remarks

We presented an attractive way of computing UIO sequences from large FSM's and identified a class of machines with guaranteed UIO's for all states. The concept of UIO tree allowed us to present a complete algorithm to compute all UIO sequences of minimal lengths. The inference rules were found to be very useful in reducing the size of a UIO tree. The projection idea was useful in avoiding the construction of any UIO tree in some cases. As an aside, Algorithm 1 computes all multiple UIO sequences for each state.

The ideas of UIO tree and inference rules were useful in proving the relation between machines with converging transitions and the existence of UIO sequences for all states. The concept of a UIO tree allowed us to prove Lemma 1, and the idea of inference rules allowed us to formulate Theorems 2 and 3.

In Section VII-B, we showed that in case all of the states of a machine have UIO sequences, even random computation of some initial UIO sequences results in significant saving in computation. The maximum essential subtrees were found to be much smaller than the complete UIO tree.

In the following, we compare the length of UIO's computed by the following indirect and direct methods: preset D sequence [3], adaptive D sequence [12], SD algorithm and Algorithm 1, and Algorithm 2.

1) UIO sequences obtained from adaptive D sequences are generally shorter than those obtained from preset D sequences.
2) However, the adaptive D sequence method does not lead to minimal length UIO sequences.
3) One can obtain minimal length UIO sequences by using the direct UIO computation techniques.
4) Since Algorithm 2 constructs a partial UIO tree and infers UIO sequences by concatenating unique prefixes, the resulting UIO sequences are generally longer than those obtained using other methods.

A disadvantage of using the inference mechanism is that UIO sequences tend to get longer. These long UIO sequences will definitely lead to longer test sequences, incurring extra cost. However, by judiciously deriving a UIO sequence for a desired state, and by using a breadth-first approach in the inference mechanism, we can reduce the impact of the inference mechanism on the length of a test sequence.

ACKNOWLEDGMENT

The author expresses his sincere thanks to the anonymous referees for their useful suggestions.

REFERENCES

[1] J. Hopcroft, "An $n \log n$ algorithm for minimizing states in a finite automaton," in *Theory of Machines and Computations,* Z. Kohavi and A. Paz, Eds. New York: Academic, 1971.

[2] A. V. Aho, J. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Reading, MA: Addison-Wesley, 1974.

[3] Z. Kohavi, *Switching and Finite Automata Theory,* 2nd ed. New York: McGraw-Hill, 1978.

[4] K. K. Sabnani and A. T. Dahbura, "A protocol test generation procedure," *Comput. Networks and ISDN Syst.,* vol. 15, no. 4, pp. 285–297, 1988.

[5] D. P. Sidhu and T. K. Leung, "Formal methods in protocol testing: A detailed study," *IEEE Trans. Software Eng.,* vol. 15, pp. 413–426, Apr. 1989.

[6] W. Y. L. Chan, S. T. Vuong, and M. R. Ito, "An improved protocol test generation procedure based on UIO's," in *ACM SIGCOMM Symp. Data Commun.,* 1989.

[7] D. Lee and M. Yannakakis, "Testing finite-state machines," in *Proc. 23rd Annu. ACM Symp. Theory of Computing,* 1991, pp. 476–485.

[8] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar, "An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese Postman Tours," *IEEE Trans. Commun.,* vol. 39, pp. 1604–1615, Nov. 1991.

[9] Y. N. Shen, F. Lombardi, and A. T. Dahbura, "Protocol conformance testing using multiple UIO sequences," *IEEE Trans. Commun.*, vol. 40, pp. 1282–1287, Aug. 1992.

[10] R. E. Miller and S. Paul, "On the generation of minimal-length conformance tests for communication protocols," *IEEE/ACM Trans. Networking,* vol. 1, pp. 116–129, Feb. 1993.

[11] H. Ural and K. Zhu, "Optimal length test sequence generation using distinguishing sequences," *IEEE/ACM Trans. Networking,* vol. 1, pp. 358–371, June 1993.

[12] D. Lee and M. Yannakakis, "Testing finite-state machines: State identification and verification," *IEEE Trans. Comput.,* vol. 43, pp. 306–320, Mar. 1994.

Kshirasagar Naik (M'94–ACM'94) received the B.S. degree from Sambalpur University, India, the M. Math. degree from the University of Waterloo, Canada, and the Ph.D. degree from Concordia University, Canada.

He is an Assistant Professor in the Deptartment of Computer Software, University of Aizu, Japan. His research interests include formal specification and testing of communication systems, mobile computing, and lightwave networks.