# Using Multiple Adaptive Distinguishing Sequences for Checking Sequence Generation

Canan Güniçen, Guy-Vincent Jourdan, Hüsnü Yenigün

# Using Multiple Adaptive Distinguishing Sequences for Checking Sequence Generation

Canan Güniçen[1], Guy-Vincent Jourdan[2], and Hüsnü Yenigün[1]

[1] Sabanci University, Istanbul, Turkey
{cananguniçen,yenigun}@sabanciuniv.edu
[2] University of Ottawa, Ottawa, Ontario, Canada
gvj@eecs.uottawa.ca

**Abstract.** A new method for constructing a checking sequence for finite state machine based testing is introduced. Unlike previous methods, which are based on state recognition using a single state identification sequence, our approach makes use of multiple state identification sequences. Using multiple state identification sequences provides an opportunity to construct shorter checking sequences, choosing greedily the state identification sequence that best suits our goal at different points during the construction of the checking sequence. We present the results of an experimental study showing that our approach produces shorter checking sequences than the previously published methods.

## 1 Introduction

Testing is an important part of the system development but it is expensive and error prone when performed manually. Therefore, there has been a significant interest in automating testing from formal specifications. Finite State Machines (FSM) are such a formal model used for specification. Deriving test sequences from FSM models, has been an attractive topic for various application domains such as sequential circuits [9], lexical analysis [1], software design [5], communication protocols [3, 6, 19, 22, 24, 25], object-oriented systems [2], and web services [13, 30]. Such techniques have also been shown to be effective in important industrial projects [11].

In order to determine whether an implementation $N$ has the same behaviour as the specification $M$, a test sequence is derived from $M$ and applied to $N$. Although, in general, observing the expected behaviour from $N$ under a test sequence does not mean that $N$ is a correct implementation of $M$, it is possible to construct a test sequence with such a guarantee under some conditions on $M$ and $N$. A test sequence with such a full fault coverage is called *a checking sequence* (CS) [23, 5].

There are many techniques that automatically generate a CS. In principle, a CS consists of three types of components: *initialization*, *state identification*, and *transition verification*. As the transition verification components are also based on identifying the starting and ending states of the transitions, a CS incorporates many applications of input sequences to identify the states of the underlying

FSM. For the state identification, we focus on the use of *Distinguishing Sequences* (DS) and in particular *Adaptive Distinguishing Sequences* (ADS) in this paper. An (A)DS does not necessarily exist for an FSM, however when it exists, it allows constructing a CS of polynomial length. Therefore many researchers have considered (A)DS based CS construction methods.

There exists a line of work to reduce the length of CS as it determines the duration and hence the cost of testing, In these works, the goal is to generate a shorter CS, by putting the pieces that need to exist in a CS together in a better way [16, 10, 29, 28, 15, 4, 26]. All of these papers focus mainly on generating as good a CS as possible for a given (A)DS, without elaborating on the choice of that (A)DS.

As a different perspective, the use of shorter ADSs is also suggested to reduce the length of a CS [27]. However an ADS provides a state identification sequence which may be short for a state but long for another state. It is thus natural to consider using several ADSs in the construction of a CS, is order to have access to a short state identification sequences for each of the states. This is the topic of this paper, in which we demonstrate that under some conditions, it is possible to use several ADSs when constructing a CS, and we experimentally show that this usually results in shorter CS than the most efficient method known so far, especially for larger FSMs. To the best of our knowledge, the only other paper paper in which using several DSs was considered was [17]. However, in that paper the goal was to overcome some problems linked to distributed, multi port systems and not to create shorter CSs. Dorofeeva et.al. also consider using multiple state identification sequences [8], but the CS construction method used in [8] is not an ADS based approach and requires the assumption that a reliable reset exists in the implementation.

In this paper, after introducing our notation and giving preliminary definitions in Section 2, we explain the motivation behind and the additional issues that need to be addressed when using multiple ADSs in CS construction in Section 3. A sufficient condition for a sequence to be a CS when a set of ADSs is used is given in Section 4. In Section 5, we first explain how we modify an existing CS generation method to use the new sufficient condition, and then present an experimental study that we performed to assess the potential improvement that can be obtained in the length of CS when multiple ADSs are used.

## 2 Preliminaries

A *deterministic finite state machine* (FSM) is specified by a tuple $M = (S, s_1, X, Y, \delta, \lambda)$, where $S$ is a finite set of states, $s_1$ is the initial state, $X$ is a finite set of input symbols, and $Y$ is a finite set of output symbols. $\delta : S \times X \rightarrow S$ is a transition function, and $\lambda : S \times X \rightarrow Y$ is an output function. Throughout the paper, we use the constants $n$, $p$, and $q$ to refer to the cardinalities $|S|$, $|X|$, and $|Y|$, respectively.

For a state $s \in S$, an input (symbol) $x \in X$, and an output (symbol) $y \in Y$, having $\delta(s, x) = s'$ and $\lambda(s, x) = y$ corresponds to a transition from the state $s$

to the state $s'$ under the input $x$ and producing the output $y$. We denote this transition by using the notation $(s, s'; x/y)$, where $s$ is called the *starting state*, $x$ is called the *input*, $s'$ is called the *ending state*, and $y$ is called the *output* of the transition.

An FSM $M$ is *completely specified* if the functions $\delta$ and $\lambda$ are defined for each $s \in S$ and for each input symbol $x \in X$. Otherwise $M$ is called *partially specified*. In this paper, we consider only completely specified FSMs.

An FSM can be depicted as a directed graph as shown in Figure 1. Here, $S = \{s_1, s_2, s_3\}$, $X = \{a, b, c\}$, and $Y = \{0, 1\}$. Each transition $(s_i, s_j; x/y)$ of the FSM is shown as an edge from $s_i$ to $s_j$ labeled by $x/y$. An FSM $M$ is called *strongly connected* if the underlying directed graph is strongly connected.

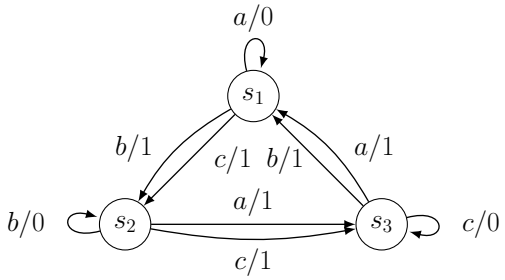


Fig. 1: The FSM $M_0$

The functions $\delta$ and $\lambda$ are extended to input sequences as explained below, where $\varepsilon$ denotes the empty sequence. For a state $s \in S$, an input sequence $\alpha \in X^\star$, and an input symbol $x \in X$, $\bar{\delta}(s, \varepsilon) = s$, $\bar{\lambda}(s, \varepsilon) = \varepsilon$, $\bar{\delta}(s, x\alpha) = \bar{\delta}(\delta(s, x), \alpha)$, and $\bar{\lambda}(s, x\alpha) = \lambda(s, x)\bar{\lambda}(\delta(s, x), \alpha)$. Throughout the paper, we will keep using the symbols $\delta$ and $\lambda$ for $\bar{\delta}$ and $\bar{\lambda}$, respectively. We also use the notation $(s_i, s_j; \alpha/\beta)$ to denote a (compound) transition from a state $s_i$ to a state $s_j$ with an input sequence $\alpha$ and an output sequence $\beta$. Note that, even though there might be more than one input symbol in $\alpha$, we still call $(s_i, s_j; \alpha/\beta)$ a transition.

Two states $s_i$ and $s_j$ of $M$ are said to be *equivalent* if, for every input sequence $\alpha \in X^\star$, $\lambda(s_i, \alpha) = \lambda(s_j, \alpha)$. If for an input sequence $\alpha$, $\lambda(s_i, \alpha) \neq \lambda(s_j, \alpha)$, then $\alpha$ is said to *distinguish* $s_i$ and $s_j$. For example, the input sequence $a$ distinguishes states $s_1$ and $s_2$ of $M_0$ given in Figure 1.

Two FSMs $M$ and $M'$ are said to be *equivalent* if the initial states of $M$ and $M'$ are equivalent. An FSM $M$ is said to be *minimal* if there is no FSM with fewer states than $M$ that is equivalent to $M$. For an FSM $M$, it is possible to compute a minimal equivalent FSM in $O(pn \lg n)$ time [18].

In this paper, we consider only deterministic, completely specified, minimal, and strongly connected finite state machines.

An *Adaptive Distinguishing Sequence* (ADS) of an FSM $M$ is a decision tree. An ADS $A$ for an FSM with $n$ states, is a rooted decision tree with $n$ leaves,

where the leaves are labeled by distinct states of $M$, internal nodes are labeled with input symbols, the edges emanating from a node are labeled with distinct output symbols. The concatenation of the labels of the internal nodes on a path from the root to the leaf labeled by a state $s_i$ is denoted by $A^i$. Note that $A^i$ is an input sequence and it is called the *State Distinguishing Sequence* (SDS) of $s_i$ in $A$. Let $Y^i$ be the concatenation of the output labels on the edges along the same root to leaf path. In this case, we also have $\lambda(s_i, A^i) = Y^i$. Since the output symbols on the edges originating from the same node are distinct, for any other state $s_j$, we have $\lambda(s_i, A^i) \neq \lambda(s_j, A^i)$. An ADS does not necessarily exist for an FSM, however the existence of an ADS can be decided in $O(pn \lg n)$ time, and if one exists, and ADS can be constructed in $O(pn^2)$ time [20].

The fault model considered in FSM based testing in the literature is in general given as follows. Let $\Phi(M)$ be the set of all FSMs with the set of input symbols $X$, with the set of output symbols $Y$, and with at most $n$ states. An implementation $N$ of an FSM $M$ is assumed to belong to the set of FSMs $\Phi(M)$. A *checking sequence* (CS) for $M$ is an input sequence that can distinguish $M$ from any faulty $N \in \Phi(M)$, that is from any $N$ which is not isomorphic to $M$.

## 3 An Illustration of the Approach

In this section, we illustrate the use of multiple ADSs for CS generation. We first provide an example to explain the advantage that using more than one ADS could provide. We then show by a counter example that one cannot simply use several ADSs while building a CS, and that some additional steps are required.

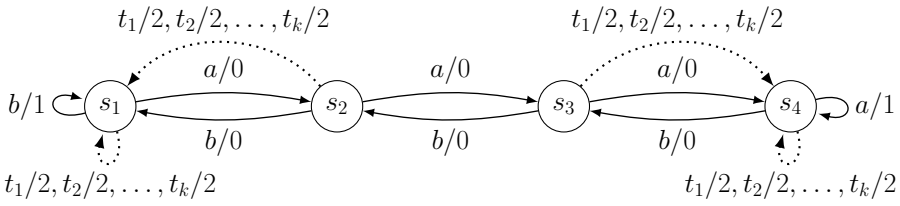


Fig. 2: The FSM $M_1$

### 3.1 A Motivational Example

The FSM $M_1$ depicted in Figure 2 has two inputs $a$ and $b$ that produce sometimes 0, sometimes 1 as output (solide edges in the picture), as well as a series of $k$ inputs $t_1, t_2, \ldots, t_k$ that always produce 2 as output (dotted edges in the picture). $M_1$ has several ADS trees, for example $A_a$ such that $A_a^1 = A_a^2 = aaa$, $A_a^3 = aa$ and $A_a^4 = a$, and $A_b$ such that $A_b^1 = b$, $A_b^2 = bb$, and $A_b^3 = A_b^4 = bbb$.

Although there are a number of different checking sequence construction methods published in the literature, all of them will require applying at least

once the SDS of the end-state of each transition after going over that transition. In our example, it means that the SDS of $s_1$ will have to be applied at least $2k+2$ times and the SDS of $s_4$ will also have to be applied at least $2k+2$ times. If one chooses to use a single ADS during the construction, say $A_a$ (resp. $A_b$), the number of inputs required for the SDS of $s_1$ is 3 (resp. 1) and the number of inputs required for the SDS of $s_4$ is 1 (resp. 3). It means that no matter which ADS is chosen, about half the time (and at least $2k+2$ times) the longest possible SDS will have to be applied.

If, in contrast, we can use *both* $A_a$ and $A_b$ when building the checking sequence, then we could use the shortest possible SDS each time, namely $A_b^1 = b$ for the transitions ending on $s_1$ and $A_a^4 = a$ for the transitions ending on $s_4$, which would result in a significant decrease in the size of the generated DS, especially if $k$ is large. We note that there are also other possible benefits coming from this choice, including more opportunities for *overlap* as well as additional choices for the ending state reached after application of the SDS in order to reduce subsequent transfer sequences. As we will see, there are however additional constraints that must be satisfied when using multiple ADSs.

### 3.2 Challenges when Using Multiple ADSs

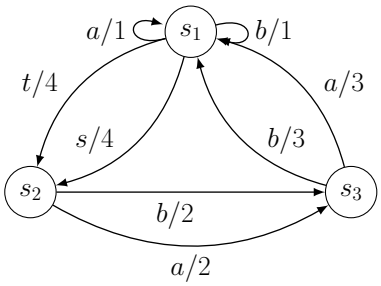


Fig. 3: The FSM $M_2$ with two ADS trees: $a$ and $b$.

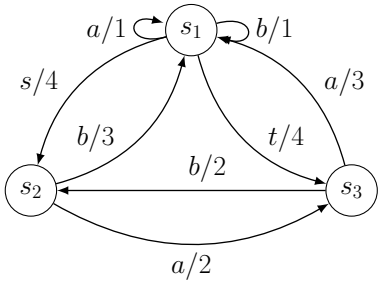


Fig. 4: This FSM is not isomorphic to the $M_2$ but produces the same output response to *aasaaabbtbbb*

In addition to the obvious problem that using multiple ADS trees requires each of these trees to be applied to every state of the FSM, another issue is what we call *Cross Verification*. In order to explain the problem, let us suppose that $A_j^i$ and $A_k^i$ are two SDSs for a state $s_i$, and they are applied to the implementation at nodes $n$ and $n'$, and the expected outputs are observed. When one considers the application of $A_j^i$ and $A_k^i$ independently, both $n$ and $n'$ are recognized as the state $s_i$. However, we cannot directly infer from the application of $A_j^i$ and $A_k^i$ that $n$ and $n'$ are actually the same implementation states. A faulty implementation may have two different states, and we might be applying $A_j^i$ and $A_k^i$ at those

states. Therefore, one needs to make sure that $n$ and $n'$ are actually the same implementation states as well. This requires some additional information to be extracted based on the observations from the implementation.

To explain the need for cross verification, suppose that we are given the FSM $M_2$ in Figure 3. We can split $M_2$ into two subgraphs as shown in Figure 5 and Figure 6, such that each subgraph has all the states of the original $FSM$ and a subset of the edges. The union of the subgraph is the original graph. Then we generate checking sequences for each subgraph, using a different ADS tree each time. We use two simple ADS trees $a$ and $b$ for subgraphs shown in Figure 5 and Figure 6, respectively. Then, we generate the checking sequences for each graph as $CS_1 = aasaaa$ and $CS_2 = bbtbbb$. Since both sequences start and end in state $s_1$, we can simply concatenate them to attempt to create a checking sequence for original FSM $M_1$, e.g. $CS_3 = aasaaabbtbbb$. Unfortunately, the resulting sequence is not a checking sequence: the FSM shown in Figure 4 produces the same output sequence as the response to $CS_3$ with the FSM of Figure 3, although it is not isomorphic to the FSM shown in Figure 3.

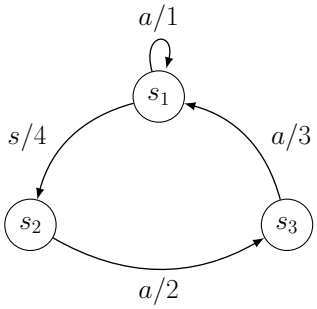


Fig. 5: A subgraph of the FSM $M_2$: $aasaaa$ is a CS

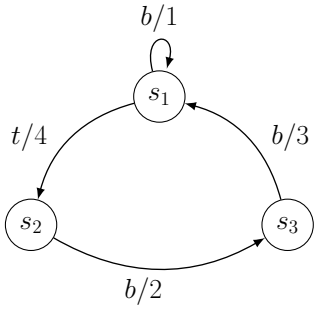


Fig. 6: Another subgraph of the FSM $M_2$: $bbtbbb$ is a CS

The problem is that although each subgraph is independently correctly verified by its own checking sequence, the states that are identified in each subgraph do not correspond to each other (in some sense, states $s_2$ and $s_3$ are swapped between the two subgraphs in this example). What we need to do, in addition to the above, is to force the fact that the node recognized by each application of the ADS in different subgraphs correspond to one another. One simple solution is to create a spanning tree on top of the original graph, and add the recognition of the spanning tree in each of the subgraphs. This way, we know that the nodes in different subgraphs correspond to the same implementation states as well. For example, if we add the spanning tree shown in Figure 7, the checking sequence for subgraph in Figure 5 doesn't change since the tree is included in it, while the checking sequence for the second subgraph in Figure 6 becomes

$CS_2$ = *bbtbbbsbab*, and the combined checking sequence is *aasaaabbtbbbsbab*, which does not produce the expected output sequence on the FSM of Figure 4.

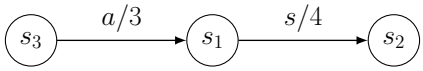


Fig. 7: Spanning tree of the FSM $M_2$

In our algorithm, we overcome this problem by differentiating between the concepts of "d-recognition" and "d-recognition by an ADS $A_j$". We declare a node d-recognized if it is d-recognized by $A_j$ for all $j$'s. This requirement forces an observation of the application of each ADS $A^j$ on the same implementation state. Such a set of observations provides information that the states recognized by different ADSs are the same implementation states. Therefore, we cross verify the node by all ADSs.

## 4 A Sufficient Condition for Checking Sequences Using Multiple ADS

For an input sequence $\omega = x_1x_2 \dots x_k$, let us consider the application of $\omega$ to an implementation FSM $N = (Q, q_1, X, Y, \delta_N, \lambda_N)$. The sequence $\alpha$ is designed as a test sequence to be applied at the state $q_1$ of $N$ that corresponds to the initial state $s_1$ of $M$. $N$ is initially assumed to be at this particular state[3]. However, since we do not know if $N$ is really at this particular state, let us refer to this unknown state of $N$ as node $n_1$. When $x_1x_2 \dots x_k$ is applied at $n_1$, $N$ will go through a sequence of states that we refer here by the node sequence $n_2n_3 \dots n_{k+1}$. Based on this sequence of nodes, we define the path $P_\omega$ as $(n_1, n_2; x_1/y_1)(n_2, n_3; x_2/y_2) \dots (n_k, n_{k+1}; x_k/y_k)$, which is the sequence of transitions of $N$ executed by the application of $\omega$. Note that $n_i$'s are the unknown states of $N$ that are traversed and $y_i$'s are the outputs produced by $N$ during the application of $\omega$. If $y_1y_2 \dots y_k \neq \lambda(s_1, \omega)$, $N$ is obviously a faulty implementation. Therefore, from now on we assume that $y_1y_2 \dots y_k = \lambda(s_1, \omega)$, and under this assumption we provide below a sufficient condition for $\omega$ to be checking sequence for $M$.

For the definitions below, let $\omega$ be an input sequence, $R$ be an equivalence relation on the set of nodes of $P_\omega$, and $\mathcal{A} = \{A_1, A_2, \dots, A_k\}$ be a set of ADSs.

Note that each node $n_i$ is a state in $N$. Based on the observations that we have in $P_\omega$, under some conditions, it is possible to infer that two different nodes $n_i$ and $n_j$ are actually the same state in $N$. We use the following equivalence relation on the nodes of $P_\omega$ to denote the set of nodes that are the same implementation state.

[3] A homing sequence or a synchronizing sequence, possibly followed by a transfer sequence is used to (supposedly) bring $N$ to this particular state.

**Definition 1.** *An equivalence relation $R$ on the nodes of $P_\omega$ is said to be an* i–equivalence *if for any two nodes $n_1, n_2$ in $P_\omega$, $(n_1, n_2) \in R$ implies $n_1$ and $n_2$ are the same (implementation) state in $N$.*

$P_\omega$ itself, when viewed as a linear sequence of application of input symbols in $\omega$, presents explicit observations on $N$. For example, having a subpath $(n_i, n_j; \alpha/\beta)$ in $P_\omega$, we have an explicit observation of the application of the input sequence $\alpha$ at node $n_i$. We call this compound transition $(n_i, n_j; \alpha/\beta)$ in $N$ an *observation*. Based on the additional information of the equivalence of the nodes in $P_\omega$, it is actually possible to *infer* some new observations that are not explicitly displayed in $P_\omega$.

**Definition 2.** *An observation $(n, n'; \alpha/\beta)$ is an* $R$–induced observation *in $P_\omega$*

*i. if $(n, n'; \alpha/\beta)$ is a subpath in $P_\omega$, or*
*ii. if there exist two $R$–induced observations $(n, n_1; \alpha_1/\beta_1)$ and $(n_2, n'; \alpha_2/\beta_2)$ in $P_\omega$ such that $(n_1, n_2) \in R$ and $\alpha/\beta = \alpha_1\alpha_2/\beta_1\beta_2$.*

*An input/output sequence $\alpha/\beta$ is said to be* $R$–observed *in $P_\omega$ at $n$ if there exists $n'$ such that $(n, n'; \alpha/\beta)$ is an $R$–induced observation in $P_\omega$.*

Note that $R$–observing an input/output sequence $\alpha/\beta$ at a node $n$ is not necessarily an explicit observation in $N$. In other words, we do not necessarily have an explicit application of the input sequence $\alpha$ at the state of $N$ represented by the node $n$ in $P_\omega$. However, when $R$ is an i–equivalence relation, it is guaranteed that if we were to apply $\alpha$ at the state of $N$ represented by the node $n$, we would have observed $\beta$. This claim is formalized below.

**Lemma 1.** *Let $R$ be an i-equivalence relation, $n$ be a node in $P_\omega$, and $\alpha/\beta$ be an input/output sequence $R$–observed at $n$. Let $s$ be the implementation state corresponding to $n$ and $\lambda_N$ be the output function of the implementation $N$. Then $\lambda_N(s, \alpha) = \beta$.*

*Proof.* The proof is immediate by induction on the length of $\alpha$, since in Definition 2 the nodes $n_1$ and $n_2$ are necessarily the same state in $N$ (when $R$ is an i–equivalence relation), and $\alpha_1$ and $\alpha_2$ are shorter than $\alpha$. □

An ADS $A_i$ of the specification $M$ is understood to be an ADS for the implementation $N$ as well, when we have the observations for the applications of all SDSs $A_i^j$ ($s_j \in S$) of $A_i$ on $N$. However, these observations do not have to be explicit observations, we can also use inferred observations.

**Definition 3.** *An ADS $A_i$ is an* $R$–valid ADS *in $P_\omega$ if for all $s_j \in S$ there exists a node $n$ in $P_\omega$ such that $A_i^j/\lambda(s_j, A_i^j)$ is $R$–observed in $P_\omega$ at $n$.*

When an ADS $A_i$ of $M$ is understood to be an ADS of $N$ as well, we can use the observations of the applications of SDSs of $A_i$ to recognize the states of $N$ as states in $M$. Definition 4 below is a generalization of d–recognition and t–recognition in the literature (see e.g. [28]) by considering that different ADSs can be used for such recognitions.

**Definition 4.** *For a node $n$ of $P_\omega$ and an ADS $A_i$, $n$ is $R$–$A_i$–*recognized as state $s_j$ *if $A_i$ is an $R$–valid ADS in $P_\omega$, and*

*i. $A_i^j/\lambda(s_j, A_i^j)$ is $R$–observed at $n$, or*
*ii. there exist nodes $n', n'', n'''$ of $P_\omega$, an ADS $A_\ell \in \mathcal{A}$, a state $s_r \in S$, and an input/output sequence $\alpha/\beta$ such that $n'$ and $n''$ are $R$–$A_\ell$–recognized as $s_r$, $n'''$ is $R$–$A_i$–recognized as $s_j$, and $(n', n; \alpha/\beta)$ and $(n'', n'''; \alpha)$ are $R$–induced observations in $P_\omega$.*

One issue that needs to be addressed when we have multiple ADSs is the following. Suppose that a node $n$ is $R$–$A_i$–recognized as state $s_j$ in $P_\omega$. Let $n'$ be another node in $P_\omega$ which is $R$–$A_k$–recognized as state $s_j$ as well, but by using another ADS $A_k$. We cannot directly deduce that $n$ and $n'$ are the same implementation states. Stated in a different way, if a node $n$ is $R$–$A_i$–recognized as state $s_j$ in $P_\omega$ for some ADS $A_i \in \mathcal{A}$, it is not necessarily $R$–$A_k$–recognized as state $s_j$ in $P_\omega$ directly for another ADS $A_k \in \mathcal{A}$. We need to have an observation of the application of $A_k^j$ at $n$ as well. Therefore, we have the following definition to make sure that a node is actually recognized as the same state $s_j$ by all the ADSs in $\mathcal{A}$.

**Definition 5.** *A node $n$ of $P_\omega$ is $R$–$\mathcal{A}$–*recognized as state $s_j$ *if for all $A_i \in \mathcal{A}$, $n$ is $R$–$A_i$–recognized as state $s_j$.*

Now we can generalize the notion of "transition verification" to the case of multiple ADSs.

**Definition 6.** *A transition $(s, s'; x/y)$ of $M$ is $R$–$\mathcal{A}$–*verified *in $P_\omega$, if there exists a subpath $(n, n'; x/y)$ in $P_\omega$ such that $n$ is $R$–$\mathcal{A}$–recognized as state $s$ and $n'$ is $R$–$\mathcal{A}$–recognized as state $s'$.*

Note that, the identity relation $\mathcal{I}$ on the nodes in $P_\omega$ is obviously an i–equivalence relation. When one uses the identity relation $\mathcal{I}$ as the relation $R$, and $\mathcal{A}$ is a singleton set, then any induced observation in $P_\omega$ must actually be a subpath of $P_\omega$. Also under this restriction, Definition 4 is equivalent to the usual state recognitions definitions (i.e. d–recognition and t–recognition given e.g. in [14]) in the literature. Therefore the following holds for this restricted case:

**Theorem 1 (adapted from Theorem 2 in [14]).** *When $\mathcal{A}$ is a singleton set, an input sequence $\omega$ is a checking sequence if all transitions of $M$ are $\mathcal{I}$–$\mathcal{A}$–verified in $P_\omega$.*

Generalizing Theorem 1 to the case where $\mathcal{A}$ is not singleton is easy.

**Theorem 2.** *Let $\mathcal{A}$ be a set of ADSs, and $\omega$ be an input sequence. If all transitions of $M$ are $\mathcal{I}$–$\mathcal{A}$–verified in $P_\omega$, then $\omega$ is a checking sequence.*

*Proof.* For each transition $(s, s'; x/y)$, by Definition 6, there exists subpath $(n, n'; x/y)$ such that $n$ and $n'$ are $\mathcal{I}$–$\mathcal{A}$–recognized as $s$ and $s'$, respectively. Based on Definition 5, $n$ and $n'$ are $\mathcal{I}$–$A_i$–recognized as $s$ and $s'$, by all $A_i \in \mathcal{A}$. Let us consider a particular ADS $A_1 \in \mathcal{A}$. The nodes $n$ and $n'$ are $\mathcal{I}$–$\{A_1\}$–recognized, and hence the transition $(s, s'; x/y)$ is $\mathcal{I}$–$\{A_1\}$–verified. Using Theorem 1, $\omega$ is a checking sequence. □

Finally we generalize the sufficiency condition to use an arbitrary i–equivalence relation $R$ as follows.

**Theorem 3.** *Let $\mathcal{A}$ be a set of ADSs, and $\omega$ be an input sequence, and $R$ be an i–equivalence relation. If all transitions of $M$ are $R$–$\mathcal{A}$–verified in $P_\omega$, then $\omega$ is a checking sequence.*

*Proof.* For each transition $(s, s'; x/y)$, by Definition 6, there exists subpath $(n, n'; x/y)$ such that $n$ and $n'$ are $R$–$\mathcal{A}$–recognized as $s$ and $s'$, respectively. Lemma 1 implies that $n$ and $n'$ are also $\mathcal{I}$–$A$–recognized as $s$ and $s'$. By Theorem 2, $\omega$ is a checking sequence. □

Theorem 3 provides a sufficient condition for a sequence to be a checking sequence. It can be used to verify if an input sequence $\omega$ is a checking sequence, provided that we are given an i–equivalence relation $R$. Lemma 2 explains how one can obtain such a relation by starting from the trivial i–equivalence relation $\mathcal{I}$, the identity relation.

**Lemma 2.** *Let $R$ be an i–equivalence relation, $n_1$ and $n_2$ be two nodes in $P_\omega$ such that $(n_1, n_2) \notin R$, and $A_i \in \mathcal{A}$ be an ADS such that both $n_1$ and $n_2$ are $R$–$A_i$–recognized as $s_j$. Consider the equivalence relation $R'$ obtained from $R$ by merging the equivalence classes of $n_1$ and $n_2$ in $R$. Then $R'$ is an i–equivalence relation.*

*Proof.* Recall that Definition 4 implies that if $n_1$ and $n_2$ are $R$–$A_i$–recognized as $s_j$, then $A_i$ is $R$–valid, meaning there are $n$ different responses $R$–observed for the application of $A_i$. If $s_1$ and $s_2$ are the implementation states corresponding to the nodes $n_1$ and $n_2$, due to the fact that $R$ is an i–equivalence relation Lemma 1 tells us that $\lambda_N(s_1, A_i^j) = \lambda_N(s_2, A_i^j)$. This is only possible when $s_1$ and $s_2$ are the same implementation states. □

Starting from the finest i–equivalence relation $\mathcal{I}$, one can use Lemma 2 repeatedly to obtain coarser i–equivalence relations. By using a coarser i–equivalence relation $R$, more $R$ induced observations will be obtained. These new inferred observations provide an opportunity to identify new i-equivalent nodes in $P_\omega$, hence to obtain an even coarser i–equivalence relation.

## 5 Experimental Study

In this section, we present an experimental study that we performed to assess the potential improvement on the length of the checking sequences that one can obtain by using multiple ADSs for checking sequence construction.

### 5.1 Checking Sequence Generation

In order to construct a checking sequence, we use a modified version of the algorithm given in [26]. The method given in [26] starts from an empty sequence $\omega = \varepsilon$, and $\omega$ is iteratively extended until it becomes a checking sequence by the sufficiency condition provided by Theorem 1. We can note here that Theorem 3 is a general sufficiency condition. Although other checking sequence construction methods can be adapted to use Theorem 3, we consider the method given in [26], since it is the most recent and a successful checking sequence construction method reported in the literature.

We modify the method to use the sufficiency condition provided by Theorem 3. Although Lemma 2 requires $R$–valid ADSs to be used when extending $R$, we found that delaying the extension of $R$ until $R$–valid ADSs are obtained is not efficient in terms of the length of the checking sequences obtained. Instead, we construct a sequence first by using Lemma 2 without requiring $R$–valid ADSs. Then, in a second phase, we extend the sequence further to force the validity of all the ADSs that have been used in the first phase.

Consider the linear path $P_\omega$ and an i-equivalence relation $R$. We keep track of a graph $G_\omega$ where each equivalence class in $R$ on the nodes of $P_\omega$ is represented by a node in $G_\omega$. An edge $(n_i, n_{i+1}; x/y)$ in $P_\omega$ is represented by an edge $([n_i], [n_{i+1}]; x/y)$ in $G_\omega$, where $[n_i]$ and $[n_{i+1}]$ are the nodes in $G_\omega$ corresponding to the equivalence classes of $n_i$ and $n_{i+1}$ in $R$. By merging the equivalence classes of $R$ into a single node in $G_\omega$, $R$–induced observations are directly represented by paths in $G_\omega$.

Similarly to the method in [26], while extending $\omega$ in each iteration, we prefer a shortest input sequence $\alpha$ to be appended to $\omega$ that can (i) recognize a state by some ADS, or (ii) perform a transition verification, or (iii) transfer to another state at which we can perform a state recognition/transition verification. The details of the method can be found in [12]. Here we simply emphasize that while deciding how to extend $\omega$, we have more alternative to chose from than the method given in [26]. First, we are using multiple ADSs and hence we need to have more state recognitions (possibly by using different SDSs). Second, if we note $n_c$ the last node in $P_\omega$, in [26] an extension of $\omega$ by an input sequence $\alpha$ is considered if there exists a node $n$ in $P_\omega$ such that $(n, n_c; \alpha'/\beta')$ is a subpath in (actually a suffix of) $P_\omega$, and $\alpha\alpha'$ is an SDS for the state corresponding to node $n$. There is a linear view used for backtracking (in order to search for overlapping opportunities) which is performed on $P_\omega$. However in our case, due to the merging of equivalence classes of the nodes of $P_\omega$ into a single node in $G_\omega$, when searching for overlapping opportunities, we backtrack from $[n_c]$ in $G_\omega$, hence we do not have a single suffix, but a tree of suffixes to chose from.

### 5.2 Selecting a Subset of ADSs

In Section 4 and Section 5.1 we explained how we generate a checking sequence when a set $\mathcal{A}$ of ADSs is given. While constructing a checking sequence, having more ADSs in $\mathcal{A}$ increases the alternatives for shorter state recognitions, hence

presents an opportunity to reduce the length of the checking sequence. However, having more ADSs in $\mathcal{A}$ also has an increasing effect on the length of the checking sequence, due to the need for the cross verification and the need to validate each ADS. Therefore, in the experiments we perform, we select a subset $\mathcal{A}^\star$ of the a given set $\mathcal{A}$ of ADSs to minimize the length of the checking sequence. The selection process is based on a greedy heuristic and it is independent on how the set $\mathcal{A}$ is constructed. Therefore, we first explain our heuristic approach to select $\mathcal{A}^\star$ in this section. The construction of the set $\mathcal{A}$ of ADSs is explained in Section 5.3.

Let $CS(M, \mathcal{A})$ be the checking sequence constructed by using the method explained in this paper, for an FSM $M$ using a set $\mathcal{A}$ of ADSs. From a given set of ADSs $\mathcal{A} = \{A_1, A_2, \ldots, A_k\}$, we start by generating a checking sequence $CS(M, \mathcal{A}')$ for each $\mathcal{A}' \subseteq \mathcal{A}$ such that $|\mathcal{A}'| = 2$. The subset $\mathcal{A}'$ giving the shortest checking sequence $CS(M, \mathcal{A}')$ is considered as the initial subset $\mathcal{A}^\star$. We then iteratively attempt to improve the length of the checking sequence by adding an ADS $A \in (\mathcal{A} \setminus \mathcal{A}^\star)$ into $\mathcal{A}^\star$. If there exists an ADS $A \in (\mathcal{A} \setminus \mathcal{A}^\star)$ such that $CS(M, \mathcal{A}^\star \cup \{A\})$ is shorter than $CS(M, \mathcal{A}^\star)$, we chose the ADS $A \in (\mathcal{A} \setminus \mathcal{A}^\star)$ such that $CS(M, \mathcal{A}^\star \cup \{A\})$ is the shortest, and update $\mathcal{A}^\star$ as $\mathcal{A}^\star = \mathcal{A}^\star \cup \{A\}$. The iterations terminate when we cannot add any ADS into $\mathcal{A}^\star$.

### 5.3 Generating a Set of ADSs

The motivation of using multiple ADSs to construct a checking sequence is that, while recognizing a state $s_i$ within a checking sequence, one can use an ADS $A_j$ such that the SDS $A_j^i$ is shorter. Therefore, it makes sense to have an ADS $A_i$ in $\mathcal{A}$ where the SDS $A_i^i$ for the state $s_i$ is as short as possible.

For an FSM with $n$ states, we start by generating an ADS $A_i$ for each state $s_i$. Therefore, we initially have at most (as some of ADSs may turn out to be the same) $n$ ADSs in $\mathcal{A}$, and we rely on the heuristic given in Section 5.2 to select a subset $\mathcal{A}^\star$. While generating the ADS $A_i$ for the state $s_i$, we aim for the minimization of the length of the SDS $A_i^i$.

Minimizing the length of the SDS of a state is introduced as MinSDS problem and it is proven to be NP–hard in [27]. Therefore a minimal length SDS $A_i^i$ is generated by considering an Answer Set Programming [21] formulation of the MinSDS problem as explained in [12]. Given an SDS $A_i^i$, we construct an ADS $A_i$ such that the SDS of state $s_i$ in $A_i$ is $A_i^i$. The details of this process can also be seen in [12], but the main idea is the following. $A_i^i$ is a path in the ADS $A_i$ that will be constructed. Let $\alpha$ be a prefix of $A_i^i$, $S'$ be the set of states not distinguished from each other by $\alpha$, distinguished from $s_i$ by $\alpha$, but not distinguished from $s_i$ by any proper prefix of $\alpha$. One can then use the standard ADS construction algorithm given in [20] to construct an ADS for the states reached from $S'$ by $\alpha$. These ADSs are used to form an ADS $A_i$ from SDS $A_i^i$.

### 5.4 Random FSM Generation

The FSMs used in the experiments are generated using the random FSM generation tool reported in [7]. For the experiments, 10 sets of FSMs are used. Each set of FSMs contains 100 FSMs having a number of states $n \in \{10, 20, \ldots, 100\}$, hence a total of 1,000 FSMs are used in the experiments. Each FSM has 5 input symbols and 5 output symbols. Under this settings, a random FSM $M$ is generated by randomly assigning $\delta(s, x)$ and $\lambda(s, x)$ for each state $s$ and for each input symbol $x$. If after this random assignment of the next states and outputs for the transition, $M$ is a strongly connected FSM with an ADS (in which case $M$ is minimal as well), then it is included in the set of FSMs to be used.

### 5.5 Experimental Results

| $n$ | $p = 5$ | $p = 9$ | $p = 13$ |
|---|---|---|---|
| 10 | 5.39 | 7.82 | 10.37 |
| 20 | 4.5 | 8.1 | 9.54 |
| 30 | 4.91 | 8.38 | 10.17 |
| 40 | 5.16 | 8.16 | 9.96 |
| 50 | 5.15 | 7.26 | 9.44 |
| 60 | 6.43 | 7.1 | 8.34 |
| 70 | 6.23 | 7.46 | 7.78 |
| 80 | 6.61 | 7.49 | 8.08 |
| 90 | 6.32 | 6.94 | 7.52 |
| 100 | 5.98 | 6.52 | 6.98 |

Table 1: Average percentage improvement in the length of checking sequences

| $n$ | $p = 5$ | $p = 9$ | $p = 13$ |
|---|---|---|---|
| 10 | 61 | 69 | 78 |
| 20 | 61 | 78 | 82 |
| 30 | 66 | 84 | 92 |
| 40 | 74 | 83 | 93 |
| 50 | 78 | 83 | 93 |
| 60 | 95 | 87 | 94 |
| 70 | 95 | 96 | 97 |
| 80 | 100 | 100 | 100 |
| 90 | 100 | 100 | 100 |
| 100 | 100 | 100 | 100 |

Table 2: Number of FSMs where $|C(M, \mathcal{A}^\star)| < |C_1|$

For an FSM $M$, let $\mathcal{A}$ be the set of ADSs computed as explained in Section 5.3. We first find the shortest checking sequence that can be generated by using a single ADS among the set $\mathcal{A}$ of ADSs. For this purpose, we compute $CS(M, \{A\})$ for each $A \in \mathcal{A}$, and find the minimum length checking sequence. Let $C_1$ be this minimum length checking sequence when a single ADS is used. For the same FSM $M$, we also compute the set $\mathcal{A}^\star$ of ADSs as explained in Section 5.2, and compute the checking sequence $C^\star = CS(M, \mathcal{A}^\star)$. The percentage improvement in the length of the checking sequence for $M$ by using multiple ADSs is then computed as $100 \times (|C_1| - |C^\star|)/|C_1|$. Note that, this improvement can be negative, when using two or more ADSs does not give a shorter checking sequence than $C_1$.

For each $n \in \{10, 20, \ldots, 100\}$, there are 100 randomly generated FSMs with $n$ states and with $p = 5$ input symbols as explained in Section 5.4. We present the average percentage improvement over 100 FSMs in Table 1. The number of FSMs in which we have a positive improvement in the length of checking sequence by using multiple ADSs is given in Table 2.

If we consider a fixed number of states for an FSM, it is expected to have a better improvement in the length of checking sequences by using multiple ADSs when there are more transitions. This is because, having the same number of states keeps the cost of cross verification constant but the savings due to the use of shorter SDSs in the transition verifications increases. In order to test this hypothesis, for each FSM $M$ that we randomly generate, we construct an FSM $M'$ (resp. $M''$) by adding 4 (resp. 8) more inputs onto $M$. The next state and the output symbols for the transitions of the additional input symbols are randomly assigned. Note however that $M'$ and $M''$ still use the same set $\mathcal{A}$ of ADSs constructed for $M$. We present the experimental results for the set of FSMs with $p = 9$ inputs and $p = 13$ inputs in Table 1 and Table 2 as well.

We would like to emphasize that $C_1$ is not a CS constructed by using a random ADS in $\mathcal{A}$, but it is constructed by using the ADS that is the best among all the ADSs in $\mathcal{A}$. Therefore the improvement figures in Table 1 are obtained against a very good ADS. We also see that, by keeping the number of states constant and increasing the number of transitions, the improvement obtained by using multiple ADSs increase as well, as hypothesized by the motivation of this work. Note that when $|C_1| < |C(M, \mathcal{A}^\star)|$, one can obviously use $C_1$ instead of $C(M, \mathcal{A}^\star)$. This approach would make the average improvement figures in Table 1 a little bit higher, since we will never have a negative improvement in this case. However the experimental results given here does not take this opportunity, and always insist on using two or more ADSs.

As the number of states increases, the percentage of FSMs in which there is an improvement in the length of the checking sequence increases, but the average improvement in the length of the checking sequence decreases. Our investigations show that, with increasing number of states, our approach starts using more ADSs in $\mathcal{A}^\star$, which pushes the cost of cross verification to higher values. For $p = 5$, our method used an average of 3 ADSs in $\mathcal{A}^\star$ for $n = 10$, whereas this average is 9 ADSs for $n = 100$. As explained in Section 5.1, our CS generation method consists of two phases, where in the second phase the sequence is basically extended to cross verify ADSs. We observe that the average length of extension in phase 2 is only 3% of the overall length of the checking sequence for $n = 10$. This percentage contribution increases with the number of states and reaches to 47% for $n = 100$.

## 6 Concluding Remarks

We presented a sufficient condition that can be used for constructing a CS using multiple ADSs. We also presented a modification of an existing CS construction method to adopt the new sufficient condition. We performed experiments to assess the potential reduction in the length of a CS that can be obtained by using multiple ADS. The experiments indicate that as the number of states increases, using multiple ADSs almost certainly decreases the length of the checking sequence, but the average improvement decreases. The investigations point to the fact that the cost of cross verification increases with the number of states.

One approach to keep the cost of cross verification limited might be to construct ADSs that has the same SDS for the states. If two ADSs $A_i$ and $A_j$ have the same SDS for a state $s_k$, then by applying this SDS at a node $n$, one would recognize $n$ as $s_k$ both by $A_i$ and $A_j$, cross verifying $A_i$ and $A_j$ at node $n$ immediately. This requires a more careful design of the set $\mathcal{A}$ of ADSs to be used in our method. Another potential improvement can come from the way the subset $\mathcal{A}^\star$ is selected. Currently, our greedy approach for selecting $\mathcal{A}^\star$ terminates when it is not possible to extend $\mathcal{A}^\star$ by adding another ADS from $\mathcal{A}$, but it does not actually mean that one cannot reduce the size of the checking sequence by using another subset of $\mathcal{A}$ with a larger cardinality than $\mathcal{A}^\star$.

As a final remark, we want to point out the fact that our improvement figures in Table 1 are obtained by comparing $CS(M, \mathcal{A}^\star)$ with $C_1 = CS(M, \{A\})$, where $A$ is the "best" ADS in $\mathcal{A}$. Note that, while constructing $C_1$, there is no need for the cross verification since there is only one ADS, but the "induced observation" idea of Definition 2 is still being used. It might be interesting to compare the length of $C_1$ by a checking sequence which is constructed by using the method given in [26] based on the same ADS $A$.

**Acknowledgment:** The authors would like to thank Robert M. Hierons and Hasan Ural for their useful input on an early version of this work.

## References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley series in computer science. Addison-Wesley Pub. Co., 1986.
2. R.V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.
3. E. Brinksma. A Theory for the Derivation of Tests. In *Proceedings of Protocol Specification, Testing, and Verification VIII*, pages 63–74, Atlantic City, 1988. North-Holland.
4. J. Chen, R.M. Hierons, H. Ural, and H. Yenigün. Eliminating Redundant Tests in a Checking Sequence. In F. Khendek and R. Dssouli, editors, *Testing of Communicating Systems*, volume 3502 of *LNCS*, pages 146–158. 2005.
5. T.S. Chow. Testing Software Design Modelled by Finite State Machines. *IEEE Transactions on Software Engineering*, 4:178–187, 1978.
6. A.T. Dahbura, K.K. Sabnani, and M.Ü. Uyar. Formal Methods for Generating Protocol Conformance Test Sequences. *Proc. of the IEEE*, 78(8):1317–1326, 1990.
7. E. Dincturk. A Two Phase Approach for Checking Sequence Generation. Master's thesis, Sabanci University, Turkey, 2009.
8. R. Dorofeeva, K. El-Fakih, and N. Yevtushenko. An improved conformance testing method. In *Formal Techniques for Networked and Distributed Systems - FORTE*, volume 3731 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2005.
9. A.D. Friedman and P.R. Menon. *Fault Detection in Digital Circuits*. Computer Applications in Electrical Engineering Series. Prentice-Hall, 1971.
10. G. Gonenc. A Method for the Design of Fault Detection Experiments. *IEEE Transactions on Computers*, 19:551–558, 1970.

11. W. Grieskamp, N. Kicillof, K. Stobie, and V.A. Braberman. Model-based Quality Assurance of Protocol Documentation: Tools and Methodology. *Software Testing, Verification and Reliability*, 21(1):55–71, 2011.
12. C. Güniçen. Checking Sequence Construction Using Multiple Adaptive Distinguishing Sequences. Master's thesis, Sabanci University, Turkey, 2015.
13. M. Haydar, A. Petrenko, and H. Sahraoui. Formal Verification of Web Applications Modeled by Communicating Automata. In *Formal Techniques for Networked and Distributed Systems (FORTE 2004)*, volume 3235 of *LNCS*, pages 115–132, Madrid, September 2004. Springer-Verlag.
14. R.M. Hierons, G.V. Jourdan, H. Ural, and H. Yenigün. Using Adaptive Distinguishing Sequences in Checking Sequence Constructions. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 682–687. ACM, 2008.
15. R.M. Hierons and H. Ural. Reduced Length Checking Sequences. *IEEE Transactions on Computers*, 51(9):1111–1117, 2002.
16. R.M. Hierons and H. Ural. Optimizing the Length of Checking Sequences. *IEEE Trans. Comput.*, 55:618–629, May 2006.
17. R.M. Hierons and H. Ural. Checking Sequences for Distributed Test Architectures. *Distributed Computing*, 21(3):223–238, 2008.
18. J.E. Hopcroft. An $n \log n$ Algorithm for Minimizing States in a Finite Automaton. Technical Report STAN-CS-71-190, Stanford University, 1971.
19. D. Lee, K.K. Sabnani, D.M. Kristol, and S. Paul. Conformance Testing of Protocols Specified as Communicating Finite State Machines-A Guided Random Walk based Approach. *Communications, IEEE Transactions on*, 44(5):631–640, 1996.
20. D. Lee and M. Yannakakis. Testing Finite-State Machines: State Identification and Verification. *IEEE Trans. Computers*, 43(3):306–320, 1994.
21. V. Lifschitz. What Is Answer Set Programming? In *AAAI*, volume 8, pages 1594–1597, 2008.
22. S.H. Low. Probabilistic Conformance Testing of Protocols with Unobservable Transitions. In *Network Protocols, 1993. Proceedings., 1993 International Conference on*, pages 368–375, 1993.
23. E.P. Moore. Gedanken-Experiments. In C. Shannon and J. McCarthy, editors, *Automata Studies*. Princeton University Press, 1956.
24. K. Sabnani and A. Dahbura. A Protocol Test Generation Procedure. *Computer Networks*, 15(4):285–297, 1988.
25. D.P. Sidhu and T.K. Leung. Formal Methods for Protocol Testing: A Detailed Study. *IEEE Transactions on Software Engineering*, 15(4):413–426, 1989.
26. A.S. Simão and A. Petrenko. Generating Checking Sequences for Partial Reduced Finite State Machines. In *Testing of Software and Communicating Systems (TestCom)*, volume 5047 of *LNCS*, pages 153–168. Springer, 2008.
27. U.C. Türker and H. Yenigün. Hardness and Inapproximability of Minimizing Adaptive Distinguishing Sequences. *Formal Methods in System Design*, 44(3):264–294, 2014.
28. H. Ural, X. Wu, and F. Zhang. On Minimizing the Lengths of Checking Sequences. *IEEE Transactions on Computers*, 46(1):93–99, 1997.
29. H. Ural and K. Zhu. Optimal Length Test Sequence Generation using Distinguishing Sequences. *IEEE/ACM Transactions on Networking*, 1(3):358–371, 1993.
30. M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-based Testing Approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.