$\mathcal{K}$-branching UIO sequences for partially specified observable non-deterministic FSMs

Khaled El-Fakih, Robert M. Hierons, *Senior Member, IEEE* and Uraz Cengiz Türker

Abstract—In black-box testing, test sequences may be constructed from systems modelled as deterministic finite-state machines (DFSMs) or, more generally, observable non-deterministic finite state machines (ONFSMs). Test sequences usually contain state identification sequences, with unique input output sequences (UIOS) often being used with DFSMs. This paper extends the notion of UIOS to ONFSMs. One challenge is that, as a result of non-determinism, the application of an input sequence can lead to exponentially many expected output sequences. To address this scalability problem, we introduce $\mathcal{K}$-UIOs: UIOS that lead to at most $\mathcal{K}$ output sequences from states of M. We show that checking $\mathcal{K}$-UIO existence is PSPACE-Complete if the problem is suitably bounded; otherwise it is in EXPSPACE and PSPACE-Hard. We provide a massively parallel algorithm for constructing $\mathcal{K}$-UIOS and the results of experiments on randomly generated and real FSM specifications. The proposed algorithm was able to construct UIOS in cases where the existing UIO generation algorithm could not and was able to construct UIOS from FSMs with 38K states and 400K transitions.

Index Terms—Software engineering/software/program verification, software engineering/testing and debugging, Software engineering/test design, Finite State Machine, Unique Input Output Sequence generation, General Purpose Graphics Processing Units.

1 Introduction

Testing is an indispensable yet costly part of software development. One promising approach to reduce development cost is to use Model Based Testing (MBT) tools and techniques. MBT methods base testing on a model of the *system under test (SUT)*. Such a model is typically defined in terms of states and transitions between states and this has led to interest in testing from a finite state machine (FSM). FSM-based approaches to MBT have been used in areas such as sequential circuits [1], lexical analysis [2], software design [3], communication protocols [3], [4], [5], [6], object-oriented systems [7], and web services [8], [9], [10], [11]. Such techniques have been shown to be effective when used in significant industrial projects [12]. Note that an FSM may represent the semantics of another model written in a more expressive language.

The literature contains many approaches that automatically generate test sequences from FSMs [6], [13], [14], [15], [16], [17], [18], [19]. For related surveys and experiments the reader may refer to [20], [21]. Let us suppose that we are testing from FSM M and, given state s of M and input sequence $\bar{x}$, let $M(s, \bar{x})$ denote the set of output sequences that M can produce if $\bar{x}$ is applied in state s. Almost all techniques for testing from FSMs use sequences that distinguish states, where $\bar{x}$ distinguishes states s and s' of M if $M(s, \bar{x})$ and $M(s', \bar{x})$ are disjoint. Such sequences are used to check the state of the SUT reached by an input sequence.

Several approaches have been developed for distinguishing the states of a deterministic FSM (DFSM). In some cases one has a distinguishing sequence, which is an input sequence that distinguishes all of the states of M, and these were used in the first FSM-based test generation algorithm [14]. However, an FSM may not have a distinguishing sequence and instead one might use a *unique input output sequence (*UIO*)* for a state s': an input sequence that distinguishes s' from all other states of M. Such a sequence need not distinguish any other pairs of states of M. Although not all FSMs have a UIO for each state, it has been reported that in practice most FSMs do have UIOS [22] and this has led to the development of many test generation methods that use UIOS [13], [16], [22], [23], [24], [25], [26], [27], [28]. Unfortunately, UIOS can be exponential long [29] and so there has been interest in methods that relatively efficiently generate UIOS [30], [31], [32], [33]. In order to increase scalability and speed up UIO derivation, Hierons and Türker introduced a massively parallel algorithm to construct UIOS from DFSMs [34]. The P-UIO algorithm scaled well; it required only a few seconds to construct UIOS from FSMs with a million states.

Traditionally, the focus has been on testing from a DFSM. However, non-determinism in a model can arise through abstraction or through the SUT being non-deterministic as a result of, for example, it being multi-threaded and there being alternative interleavings. This has led to interest in non-deterministic FSMs and, in particular, to *observable non-deterministic FSMs (ONFSMs)*; those where for every state s, input x and output y there is at most one transition from s that has input x and output y [35], [36]. A number of authors have explored distinguishing sequences for ONFSMs [29], [37], [38], [39], [40], [41]. However, it appears that there are no published scalable algorithms for generating UIOS from ONFSMs. Naturally, there is a need to generalise the concept of a UIO to ONFSMs and algorithms that generate UIOS from DFSMs cannot be directly used.

Due to non-determinism, the number of possible output sequences that can result from applying input sequence $\bar{x}$ to state s of ONFSM M can grow exponentially with the

- *Author names are given according to the Hardy-Littlewood Rule, i.e., author names are provided in alphabetical order regarding the family names in ascending order.*

length of $\bar{x}$. Since we are interested in algorithms that scale to large ONFSMs, this paper introduces the notion of a $\mathcal{K}$-branching UIO ($\mathcal{K}$-UIO): a UIO $\bar{x}$ that can result in at most $\mathcal{K}$ output sequences (when applied in a state s' of M). We prove that the problem of deciding whether state s of ONFSM M has a $\mathcal{K}$-UIO is in EXPSPACE and is PSPACE-hard; if we bound $\mathcal{K}$ by a polynomial in terms of the size of M then the problem is PSPACE-complete. We also extend the notion of a unique predecessor, used by Naik [33] when generating UIOs from DFSMs, to $\mathcal{K}$-UIOs and ONFSMs. We then introduce a parallel $\mathcal{K}$-UIO generation algorithm for use on GPUs and evaluate it by comparing it against a breadth-first algorithm.

This paper is organised as follows. Section 2 defines FSMs and associated notation/terminology. Section 3 introduces $\mathcal{K}$-UIOs and generalises unique predecessors. Section 4 investigates the complexity of deciding whether an ONFSM has a $\mathcal{K}$-UIO for state s. In Sections 5 and 6 we outline the new algorithm for generating $\mathcal{K}$-UIOs from an ONFSM. Section 7 describes the experiments carried out and, finally, Section 8 draws conclusions and describes possible lines of future work.

2 Preliminaries

Definition 2.1. A non-deterministic FSM (NFSM) is defined by a tuple $M = (S, s_0, X, Y, h)$ where: $S = \{s_1, s_2, \ldots, s_n\}$ is a finite set of states, $s_0 \in S$ is the initial state, $X = \{x_1, x_2, \ldots, x_r\}$ is the finite set of inputs, $Y = \{y_1, y_2, \ldots, y_v\}$ is the finite set of outputs, and $h \subseteq S \times X \times Y \times S$ is the set of transitions. We assume that X is disjoint from Y.

Tuple $\tau = (s, x, y, s') \in h$ is a *transition* of M and has *starting state* s, *ending state* s', and *label* x/y. The label x/y has *input portion* x and *output portion* y. We can interpret τ as meaning that if M receives input x when in state s then it can output y and move to state s'. M can then receive another input when in state s'.

We use ε to denote the empty sequence and given sequences $\bar{x}$ and $\bar{x}'$, $\bar{x}\bar{x}'$ denotes the concatenation of $\bar{x}$ and $\bar{x}'$. Given input/output pairs $x_1/y_1, \ldots, x_k/y_k$, $x_1/y_1 \ldots x_k/y_k$ and also $x_1x_2\ldots x_k/y_1y_2\ldots y_k$ denote the corresponding input/output sequence (or *trace*) σ: the sequence that starts with x_1/y_1, then ... and finally x_k/y_k. Further, $i(\sigma) = x_1 \ldots x_k$ and $o(\sigma) = y_1 \ldots y_k$ denote the *input and output portions* respectively of σ.

An NFSM is an observable NFSM (ONFSM) if for all $s \in S$, $x \in X$, and $y \in Y$ there is at most one state $s' \in S$ such that $(s, x, y, s') \in h$. Throughout this paper $M = (S, s_0, X, Y, h)$ will denote an ONFSM from which we are testing and we use 'FSM' to denote such machines. An input x is defined in state s if there exists some s' and y such that $(s, x, y, s') \in h$. We allow *partial* FSMs: an input x might not be defined in some state s; if an FSM is not partial then it is completely-specified.

An FSM can be represented by a directed graph. Figure 1 represent an FSM M_1 with state set $\{s_1, s_2, s_3, s_4\}$, input set $\{x\}$, and output set $\{y_1, y_2, y_3\}$. A node represents a state and a directed edge with label x/y, from a node with label s to a node with label s', denotes transition $\tau = (s, x, y, s')$.

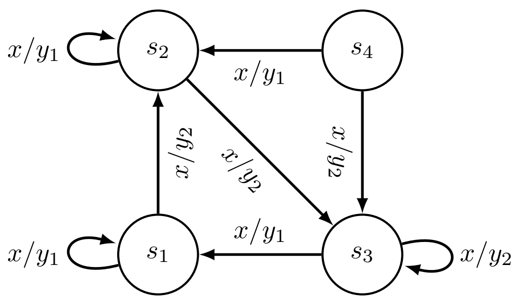


Figure 1: Example FSM M_1.

The behaviour of an FSM M is defined in terms of the labels of walks leaving the initial state; such labels of walks are called *traces*. A *walk* ρ of M is a sequence $(s_1, x_1, y_1, s_2)(s_2, x_2, y_2, s_3) \ldots (s_k, x_k, y_k, s_{k+1})$ of consecutive transitions and ρ has starting state s_1, ending state s_{k+1}, and label $x_1/y_1 \ldots x_k/y_k$. For example, $\rho_1 = (s_1, x, y_1, s_1)(s_1, x, y_2, s_2)(s_2, x, y_2, s_3)$ is a walk of M_1; ρ_1 has *starting state* s_1, *ending state* s_3, and *label* $x/y_1\, x/y_2\, x/y_2$. Here $x/y_1\, x/y_2\, x/y_2$ is a *trace* of M_1.

We can extend the notion of an input being defined in state s to input sequences as follows.

Definition 2.2. Input sequence $\bar{x}$ is defined in state s of M if either $\bar{x} = \varepsilon$ or $\bar{x} = \bar{x}'x$ for input sequence $\bar{x}'$ and input x such that the following conditions hold:

1) $\bar{x}'$ is defined in state s; and
2) x is defined in every state s' of M that is the ending state of a walk ρ of M that has starting state s and a label whose input portion is $\bar{x}'$.

FSM M defines the language $L(M)$ of labels of walks with starting state s_0 and $L_M(s)$ denotes the language obtained if we make s the initial state. Thus, $L_M(s) = \{x_1 \ldots x_m/y_1 \ldots y_m \in X^*/Y^* | \exists s_1, \ldots, s_{m+1}. s_1 = s \wedge \forall 1 \leq i \leq m. (s_i, x_i, y_i, s_{i+1}) \in h\}$. For example, $x/y_1x/y_2 \in L_{M_1}(s_1)$. Given $S' \subseteq S$, $L_M(S') = \cup_{s \in S'} L_M(s)$ is the set of traces that can be produced if the initial state of M is in S'. Given state s and input sequence $\bar{x}$ we use $M(s, \bar{x}) = \{\sigma \in L_M(s) | i(\sigma) = \bar{x}\}$ to denote the set of traces in $L_M(s)$ that have input portion $\bar{x}$. Given state set $S' \subseteq S$, $M(S', \bar{x}) = \cup_{s \in S'} M(s, \bar{x})$ denotes the set of traces that can result from applying $\bar{x}$ to a state in S'.

States s, s' of M are *equivalent* if $L_M(s) = L_M(s')$ and FSMs M and N are *equivalent* if $L(M) = L(N)$. FSM M is *minimal* if there is no equivalent FSM with fewer states. FSM M is *strongly connected* if for every ordered pair (s, s') of states, M has a walk with starting state s and ending state s'. As usual, in this paper we consider only minimal FSMs. This is not a restriction since an (observable) FSM can be rewritten to an equivalent minimal FSM in polynomial time using any technique that minimises a deterministic finite automaton.

3 $\mathcal{K}$-UIOs and unique-predecessors

In this section, we initially define $\mathcal{K}$-UIOs and we then generalise the notion of a unique predecessor [33].

Definition 3.1. Given FSM M, an input sequence $\bar{x}$ is a *unique input output sequence* for state s of M if $\bar{x}$ is defined in all states of M and $M(s,\bar{x}) \cap M(S \setminus \{s\}, \bar{x}) = \emptyset$.

As discussed in Section 1, during testing we may prefer to have UIOs that lead to a limited number of traces (say $\mathcal{K}$ traces) and now we define what it means a UIO to be a $\mathcal{K}$-branching UIO sequence.

Definition 3.2. Given FSM M, input sequence $\bar{x}$ is a $\mathcal{K}$-*branching* UIO *(*$\mathcal{K}$-UIO*)* for state s of M if $\bar{x}$ is a UIO for s and $|M(s',\bar{x})| \leq \mathcal{K}$ for all $s' \in S$.

We now define the problem investigated in this paper.

Definition 3.3. The $\mathcal{K}$-UIO problem is to decide whether state s of an ONFSM has a $\mathcal{K}$-UIO.

Naik [33] introduced unique-predecessors for use in generating UIOs for a DFSM. Given transition (s,x,y,s') of DFSM D, s is a *unique-predecessor* of s' if (s,x,y,s') is the only transition of D that has input x, output y, and ending state s'. This has the following useful property: if $\bar{x}$ is a UIO for s' then $x\bar{x}$ is a UIO for s. As a result, once a UIO has been found for a state, it may be possible to use unique-predecessors to generate UIOs for other states of D. The following definition is provided in [33] and we use the term det-unique predecessor to distinguish it from the generalised notion below.

Definition 3.4. Let us suppose that D is a DFSM with state set S. Given state $s \in S$, state $s' \in S \setminus \{s\}$ is a *det-unique predecessor* for s if there exists a transition τ with $start(\tau) = s'$, $end(\tau) = s$ and $label(\tau) = x/y$ such that there exists no transition $\tau' \neq \tau$ with $end(\tau) = s$ and $label(\tau) = x/y$.

We would like to define a similar notion for FSMs and $\mathcal{K}$-UIOs but this is complicated by non-determinism and the need to bound the size of the $M(s',\bar{x})$. The following generalisation allows non-determinism but, in using this, we will require that an input sequence $\bar{x}$ is a UIO for all states in a set S' (the states reached by x).

Definition 3.5. Given $S' \subseteq S$, state s is a unique predecessor of S' through input x if the following hold.

1) $S' = \{s'' \in S | \exists y \in Y.(s,x,y,s'') \in h\}$.
2) For all $s_1 \in S'$ and $y \in Y$ such that $(s,x,y,s_1) \in h$, there does not exist $s_2 \in S \setminus \{s\}$ such that $(s_2,x,y,s_1) \in h$.

The first condition requires that S' is the set of states reached from s by x. Let us suppose that we apply x in state s, observe output y, and M moves to $s' \in S'$. The second condition of Definition 3.5 tells us that in order to show that we started in s it is sufficient to show that s' was the state reached by x. The following is clear.

Proposition 3.1. If s is an S' unique predecessor through input x, and for all $s' \in S'$ we have that $\bar{x}$ is a UIO for s', then $x\bar{x}$ is a UIO for s.

Let us suppose that s is a unique predecessor of S' through x and this property of x is to be used in generating $\mathcal{K}$-UIOs. Further, let us suppose that $\bar{x}$ is a $\mathcal{K}$-UIO for all states in S'. We need to place an upper bound on the number of output sequences that can result from applying $x\bar{x}$ in a state of M. Such an output sequence is formed from an output in response to x and then an output sequence in response to $\bar{x}$. We are thus interested in the following value, which we call the output branching degree of x: $obd_M(x) = max_{s \in S}|\{y \in Y | \exists s_1 \in S.(s,x,y,s_1) \in h\}|$.

We can now show how unique predecessors can be used to generate additional $\mathcal{K}'$-UIOs.

Proposition 3.2. If s is an S' unique predecessor through input x, and for all $s' \in S'$ we have that $\bar{x}$ is a $\mathcal{K}$-branching UIO for s', then $x\bar{x}$ is a $\mathcal{K}'$-branching UIO for s where $\mathcal{K}' = obd_M(x) \cdot \mathcal{K}$.

4 Complexity of the $\mathcal{K}$-UIO problem

In this section, we consider the computational complexity of the $\mathcal{K}$-UIO problem. First we derive an upper bound on the length of a shortest $\mathcal{K}$-UIO for a state s of an FSM M. This upper bound will be used in reasoning about the complexity of deriving $\mathcal{K}$-UIOs and we do not claim that it is a tight upper bound.

In the following, given a state s of M we let $Walks(s)$ denote the set of walks of M that have starting state s and given a walk ρ we let $label(\rho)$ denote the label of ρ and $end(\rho)$ denote the end state of ρ.

Lemma 4.1. A state s of an FSM M has a $\mathcal{K}$-UIO if and only if it has a $\mathcal{K}$-UIO of length no greater than $n^{n\mathcal{K}}2^{n\mathcal{K}}$.

Proof: Let us suppose that $\bar{x} = x_1 \dots x_m$ is a shortest $\mathcal{K}$-UIO for s. Let $R = M(s,\bar{x}) = \{\sigma_1, \dots, \sigma_k\}$ so $k \leq \mathcal{K}$.

For $\sigma_i \in R$ and $1 \leq j \leq m$ let $pre(\sigma_i, j)$ denote the prefix of σ_i of length j. Further, we will let $s(i,j)$ denote the unique state of M reached from s by a walk with label $pre(\sigma_i, j)$ and let $C(i,j)$ denote the set of states reached from states in $S \setminus \{s\}$ by walks with label $pre(\sigma_i, j)$. We therefore have that $s(i,j)$ is the unique state of M such that there exists $\rho \in Walks(s)$ with $label(\rho) = pre(\sigma_i, j)$ and $end(\rho) = s(i,j)$. In addition,

$$C(i,j) = \left\{ \begin{array}{l} s' \in S | \exists s_1 \in S \setminus \{s\}, \rho \in Walks(s_1). \\ label(\rho) = pre(\sigma_i, j) \wedge s' = end(\rho) \end{array} \right\}$$

Given $s' \in S \setminus \{s\}$ let $D(s',j)$ denote the tuple of ending states of walks from s' whose label is in $M(s', x_1 \dots x_j) \setminus M(s, x_1 \dots x_j)$. $D(s',j)$ is a tuple, rather than a set, since we wish to distinguish between two different walks that reach the same state (we use the $D(s',j)$ to check that the bound $\mathcal{K}$ is respected).

Now consider some $1 \leq j < \ell < m$ and prefixes $x_1 \dots x_j$ and $x_1 \dots x_\ell$ of $\bar{x}$. Since $\bar{x}$ is a $\mathcal{K}$-UIO, $x_{\ell+1} \dots x_m$ distinguishes $s(i,\ell)$ from all states in $C(i,\ell)$. Thus, if for all $1 \leq i \leq k$ we have that $s(i,j) = s(i,\ell)$ and $C(i,j) = C(i,\ell)$ then $x_1 \dots x_j x_{\ell+1} \dots x_m$ is a UIO for s. In addition, if $D(s',j) = D(s',\ell)$ for all $s' \in S \setminus \{s\}$ then $|M(s'', x_1 \dots x_j x_{\ell+1} \dots x_m)| \leq |M(s'', x_1 \dots x_m)|$ for all $s'' \in S$ and so $x_1 \dots x_j x_{\ell+1} \dots x_m$ is a $\mathcal{K}$-UIO for s. This contradicts the minimality of $\bar{x}$. Thus, for all $1 \leq j < \ell < m$ there is some i such that either $s(i,\ell) \neq s(i,j)$, $C(i,\ell) \neq C(i,j)$, or there exists $s' \in S \setminus \{s\}$ such that $D(s',j) \neq D(s',\ell)$. The length of $\bar{x}$ is thus bounded above by the number of possible values for the $D(s',j)$ and $(s(1,j), C(1,j), s(2,j), C(2,j), \dots, s(k,j), C(j,k))$.

Given $1 \leq i \leq k$, there are n possible values for $s(i,j)$ and no more than 2^n (the number of subsets of S) values for $C(i,j)$. Each $D(s',j)$ contains at most $\mathcal{K}$ states and so there are at most $n^{\mathcal{K}}$ values for a $D(s',j)$ and at most $n^{(n-1)\mathcal{K}}$ values for the set of $D(s',j)$. Thus, there are at most $(n2^n)^k n^{(n-1)\mathcal{K}} = n^{k+(n-1)\mathcal{K}} 2^{nk}$ possible values of $(s(1,j), C(1,j), s(2,j), \ldots, s(k,j), C(j,k))$ and the $D(s',j)$. The result follows from $k \leq \mathcal{K}$. □

We can now reason about the space a non-deterministic Turning Machine might use.

Proposition 4.1. It is possible for a non-deterministic Turing Machine to decide whether a state s of M has a $\mathcal{K}$-UIO in $O(n\mathcal{K}\log n)$ space.

Proof: We will describe such a non-deterministic Turing Machine and this will guess one input at a time. It will maintain a list of tuples of the form (s', C) $(C \subseteq S)$ such that if the current guessed input sequence is $\bar{x} = x_1 \ldots x_j$ then there is a possible trace $\sigma \in M(s, \bar{x})$ such that if M started in s and σ has been observed then M must now be in s' and C is the set of states that M might be in if σ has been observed and M did not start in s. More formally, there is a walk $\rho \in Walks(s)$ with $s' = end(\rho)$ and $\sigma = label(\rho)$ and $C = \{s_2 \in S | \exists s_1 \in (S \setminus \{s\}), \rho' \in Walks(s_1).label(\rho') = \sigma \wedge s_2 = end(\rho')\}$. A tuple (s', C) can be stored in $O(n)$ space. In each iteration, a next input is guessed and the tuples are updated in the natural way. If the number of tuples exceeds $\mathcal{K}$ then the process terminates with failure; if all of the C are empty then a $\mathcal{K}$-UIO has been found. The space required to store the tuples is of $O(n\mathcal{K})$.

In addition, for each state $s' \in S \setminus \{s\}$, the Turing machine records the tuple $P(s')$ of states of M reached from s' by walks whose label has input portion $\bar{x}$. As before, $P(s')$ is a tuple and not a set (we wish to know the number of walks that reach each state). The computation terminates with failure if the size of a $P(s')$ exceeds $\mathcal{K}$ and so the set of $P(s')$ take at most $n\mathcal{K}$ space.

We add counter c to count the number of iterations; the Turing Machine terminates with failure if the counter exceeds bound $n^{n\mathcal{K}} 2^{n\mathcal{K}}$ (Lemma 4.1). This counter takes $O(log(n^{n\mathcal{K}} 2^{n\mathcal{K}})) = O(n\mathcal{K}\log n)$ space. Thus, the overall space is of $O(n\mathcal{K}\log n)$. □

The space requirements are bounded above by a polynomial in n and $\mathcal{K}$. However, the complexity is exponential in terms of the size of the representation of $\mathcal{K}$, which takes $O(\log \mathcal{K})$ space. We obtain the following.

Theorem 4.1. The problem of deciding whether a state has a $\mathcal{K}$-UIO is in EXPSPACE and is PSPACE-hard.

Proof: First, by Proposition 4.1 we know that a non-deterministic Turing Machine can solve the problem in exponential space. Since non-deterministic EXPSPACE is equal to deterministic EXPSPACE [42], we have that the problem is in EXPSPACE.

We now show that the problem is PSPACE-hard. Consider the case where M is deterministic and we have that $\mathcal{K} = 1$. Then there is a $\mathcal{K}$-UIO for state s of M if and only if there is a UIO for s. Thus, any algorithm that can decide whether a state s of M has a $\mathcal{K}$-UIO can also be used to decide whether a state of a DFSM has a UIO. Since the problem of deciding whether a state of a DFSM has a UIO is PSPACE-hard, we have that the problem of deciding whether a state s of M has a $\mathcal{K}$-UIO is also PSPACE-hard. The result thus follows. □

In practice, we are likely to restrict $\mathcal{K}$ to being relatively small, with this motivating the following result.

Theorem 4.2. If $\mathcal{K}$ is bounded above by a polynomial in terms of n then the problem of deciding whether a state s of M has a $\mathcal{K}$-UIO is PSPACE-complete.

Proof: First, by Proposition 4.1 we know that a non-deterministic Turing Machine can solve the problem in $O(n\mathcal{K}\log n)$ space. Since $\mathcal{K}$ is bounded above by a polynomial in terms of n, there is a polynomial upper bound on the space and so the problem is in PSPACE.

The problem being PSPACE-hard follows in the same manner as the proof of Theorem 4.1 □

5 Overview of the Hi-DFS algorithm

In this section, we describe how ideas from GPU Computing were used to devise a novel algorithm (Hi-DFS) for generating $\mathcal{K}$-UIOs from an FSM. We start by describing the intuition behind the proposed algorithm and then we explain the algorithm in detail.

To compute $\mathcal{K}$-UIOs, it is important to reduce the space requirements to support scalability; to provide an efficient, GPU friendly representation for the data; and to use effective, parallelisable methods. These observations led us to introduce a novel *hash-based iterative depth-first-search* approach to find $\mathcal{K}$-UIOs.

The naïve approach to derive UIOs from an FSM, is to use breadth first search to construct a tree structure (`successor tree`) [33]. For every node of such a tree, there are at most $|X||Y|$ outgoing edges and so there may be exponentially many nodes in the tree, which can lead to the usage of very large memory space and hence can reduce the scalability of the underlying algorithm.

In order not to construct such a tree, recently Hierons and Türker [34] proposed a parallel UIO generation algorithm for DFSMs. In their algorithm, a UIO for a state is computed by sorting outputs produced by the states of the FSM as a response to an input sequence. Note that for DFSMs, if an input sequence is applied to a state, we may observe only one output sequence that takes the FSM to exactly one state. Thus, after sorting, by checking the neighbouring output sequences, one can state whether an output sequence is unique. If it is unique then we can declare that the associated state is distinguished from others [35]. However, the above approach cannot be used with ONFSMs since we may observe more than one output sequence from a state. To handle this problem, we use a *hash* based approach to reveal distinguished states.

The proposed algorithm receives an ONFSM M and three integers B, D and $\mathcal{K}$ as its inputs. Then the algorithm executes the following steps.

Step 1) Extracting unique predecessor information: The algorithm investigates the transition structure of the underlying FSM to reveal unique predecessors.

Step 2) Generating input sequences: The Hi-DFS algorithm searches for $\mathcal{K}$-UIOs using Depth First Search (DFS). To do this, the algorithm generates B different input sequences of length D. The input sequences to be applied during DFS are generated in such a way that throughout the execution the same input sequence cannot be applied to set S more than once and if the algorithm uses an input sequence $\bar{x}$ then $\bar{x}$ is applied to all states of the FSM (to check whether $\bar{x}$ is a $\mathcal{K}$-UIO).

Step 3) Applying DFS: The Hi-DFS algorithm performs a number of (maximum is B) DFSs to the states of M in parallel. To do so, we use $\mathbb{S}$ to denote the multi-set of copies of the states of the underlying FSM i.e., $\mathbb{S}$ contains multiple copies of S; these will be processed in parallel and we have that $|\mathbb{S}| \leq B$. The essential idea is that we apply different input sequences to the sets of states of M in parallel. To do so, the algorithm applies generated inputs to the sets of states in multi-set $\mathbb{S}$. While doing this, it stores the outputs and reached states (i.e., current states). The algorithm stops applying a DFS, if the number of traces from a state s exceeds $\mathcal{K}$ or there exists no defined transition to which to apply DFS.

Step 4) Gather the result of DFS: The algorithm may collect multiple (at most $\mathcal{K}$) traces from a state. When determining whether a state has been distinguished, we need to compare its traces with the traces from other states. Following a naïve approach that relies on a successor tree, we need to compare $\mathcal{K}$ traces of length D for s to $\mathcal{K}$ traces of length D for all states (n). This requires $O(n\mathcal{K}^2D)$ steps for a single state.

In this paper, we introduce a novel massively parallel hash based method to check whether a state is distinguished. If there are enough computing cores then the time required to reveal distinguishing information for all states in S is $O(n\mathcal{K}Dlog(n\mathcal{K}D))$. We describe the method in the next section. If a $\mathcal{K}$-UIO has been computed, by using the predecessor information, the algorithm searches for additional UIOs.

Step 5) Deciding next step: When the algorithm finishes analysing the outcome of DFSs, the algorithm will follow one of the following steps:

- Continue from step 2: if not all $\mathcal{K}$-UIOs are found and not all input sequences have been derived yet.
- Terminate otherwise.

Please see Algorithm 1 for details.

6 Low level description of Hi-DFS

The proposed algorithm first generates unique predecessor information. Afterwards it starts to generate $\mathcal{K}$-UIOs. We will use the FSM and the input parameters in Figure 2 to show how the Hi-DFS algorithm works.

6.1 Constructing unique predecessors information

Unique predecessor information is constructed in two steps. First, for each state s and input-output pair x/y, we count the number of transitions that end in s and have label x/y, recording this information in a *Counter Vector* CV. The basic idea is that for state s, the counter vector has a vector (an *input/output vector (IOV)*) $CV(s) = re$ of integers. This IOV re has size $|X||Y|$ and the element of re with index x/y holds the number of transitions that end at s and have label x/y.

Algorithm 1: $\mathcal{K}$-UIO generation algorithm, highlighted instructions are executed by GPU.

```
Input: FSM $M$ with $S, X$ and $Y$, positive integers $B, \mathcal{K}$ and $D$
Output: A set ($\mathcal{U}$) of $\mathcal{K}$-UIOs for $M$
begin
1    Initialize counter ($CV$), input output ($IOV$) vectors, unique
     predecessor table ($UPT$) and set of $\mathcal{K}$-UIOs $\mathcal{U}$
2    Create a forest vector element $e$ and initialize it as $e_S = S$ and
     $\Gamma = \emptyset$
3    Initialize a forest vector $FV \leftarrow e$
4    UniquePredecessorGenKernel($M, CV, IOV, UPT$)
5    while true do
6        $\Omega \leftarrow$GetInputs($B, D, FV$) such that $b \leftarrow |\Omega|$, and $b \leq B$
7        if All states have $\mathcal{K}$-UIOs or $b = 0$ then
8            Return $\mathcal{U}$
9        Create a block vector $\mathcal{B} \leftarrow \{\Delta_1, \Delta_2, \dots, \Delta_b\}$
10       $i \leftarrow 0$
11       DFSKernel($\mathcal{B}, D, \mathcal{K}$)
12       foreach $\Delta$ of $\mathcal{B}$ do
13           $\Psi_\Delta \leftarrow \{\bar{\chi}_{r1}, \bar{\chi}_{r2}, \dots, \bar{\chi}_{rn}\}$
14           Sort($\Psi_\Delta$)
15           $\bar{\mathcal{U}} \leftarrow$Hash($\Psi_\Delta$)
16           UniquePredecessorKernel($\bar{\mathcal{U}}, UPT$)
17           $\mathcal{U} \leftarrow \mathcal{U} \cup \bar{\mathcal{U}}$
```

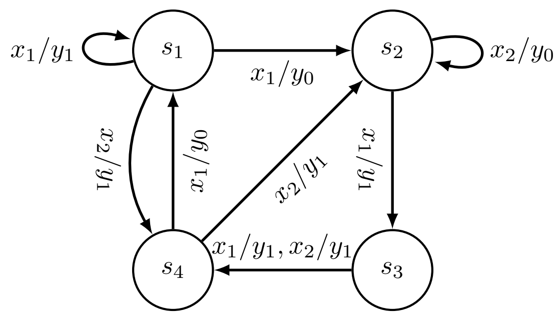


Figure 2: Example FSM M_2. Inputs to the algorithm are: $B = 3$, $D = 2$, and $\mathcal{K} = 2$.

The CV is created by the *unique predecessor kernel* (Line 1 of Algorithm 1). This kernel has one thread for each transition and the thread associated with transition (s, x, y, s') increases (by one) the value that records the number of transitions that have label x/y and ending state s'. This is achieved as follows. In the unique predecessor kernel, for each transition τ_i, we introduce a single thread t_i. Let x/y be the label and s and s' the starting and ending states of τ_i, respectively. If we use CV to denote the counter vector then thread t_i increments the element of $CV(s')$, that has index x/y, by one (note that to prevent race conditions, this addition must be serial). Thus, for each transition τ_i that ends at state s' and has label x/y, exactly one thread increases the element of $CV(s')$ that has index x/y. As a result, at the end, the element of $CV(s')$ that has index x/y records the number of transitions with label x/y that end at state s'.

In the second step, we use the counter vector CV to fill a table called the *unique predecessor table*.

Definition 6.1. A *unique predecessor table* (UPT) is a table that records the combinations of starting state $s \in S$, input/output pair x/y, and ending state $s' \in S$ such (s, x, y, s') is a transition of M and there is no other transition of M with ending state s' and label x/y.

After all threads have updated the IOV, threads check

Reached state	s_1				s_2				s_3				s_4			
I	x_1		x_2		x_1		x_2		x_1		x_2		x_1		x_2	
O	y_0	y_1	y_0	y_1	y_0	y_1	y_0	y_1	y_0	y_1	y_0	y_1	y_0	y_1	y_0	y_1
Unique predecessor	s_4	s_1	—	—	s_1	—	—	s_4	—	s_2	—	—	—	s_3	—	—

Table 1: The UPT constructed by the Hi-DFS algorithm for M_2.

the values in re (using a loop that iterates $|X||Y|$ times). If for transition τ that starts at s and ends in s' the re value is 1 then s is a unique predecessor for s' and τ is recorded in the UPT. Note that for a state s there can be at most $(n-1)|X||Y|$ different unique predecessors and so the size of the UPT is bounded above by $n^2|X||Y|$. Table 1 has the UPT computed for M_2.

6.2 Generating $\mathcal{K}$-UIOs

6.2.1 Generating input sequences

Input sequences are generated on the CPU (Line 6 of Algorithm 1). To perform this step, we keep a vector structure (forest vector) that keeps a forest of *input trees*.

Definition 6.2. An input tree Φ is a tree where each node ρ of Φ labels an element $\rho(i)$ from X such that children of the root of a subtree Φ' must have different labels (and so there may be at most $|X|$ such children). A subtree is *full*, if it has exactly $|X|$ children; otherwise it is *partial*. (Please see Figure 3 for examples.)

We now define the forest vector (FV).

Definition 6.3. A forest vector (FV) for an FSM is a vector of elements where each element is associated with an ordered set of input trees $\Gamma = \{\Phi_1, \Phi_2, \ldots\}$ with different root labels, where ordering is done according to the root labels i.e., Φ_i precedes Φ_j if $\rho(i) < \rho(j)$. For M_2, the Hi-DFS algorithm generates the FV in Figure 3.

Let $\mathbb{S} = \{S'_0, S'_1 \ldots S'_B\}$ be a set of sets of states, then for each set S' in set $\mathbb{S}$, the Hi-DFS algorithm picks an input sequence by evaluating the resultant forest Γ: while the input sequence is of length less than D, the algorithm randomly picks a partial input tree Φ and finds the root (ρ) of a shallowest partial sub-tree whose depth is smaller than D. Afterwards it introduces a child node ρ' and labels it with an input x such that ρ does not have a child labeled by x. If the depth does not reach D (note that the algorithm should return an input sequence of length D), the algorithm repeats this step i.e., it introduces a child node ρ'' to the node ρ'. This is followed by concatenating the labels of the nodes on the path from the root of the input tree to the lastly added fresh node to generate an input sequence w.

As this structure holds the input sequences to be applied, the memory requirement for a set of states is $O(|X|^D)$, where D is the depth of the DFS.

By the definition and construction of input trees it should be clear that the Hi-DFS algorithm exhaustively generates input sequences of length D.

Lemma 6.1. Given an FSM M, integers B, D, and $\mathcal{K}$, and a set of states S', the Hi-DFS algorithm generates all input sequences of length D for S'.

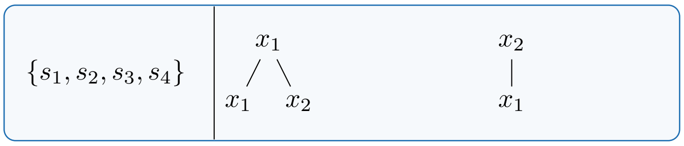


Figure 3: An FV constructed during the execution of Hi-DFS for M_2. Generated input sequences are $w_1 = x_2x_1, w_2 = x_1x_1$ and $w_3 = x_1x_2$.

6.2.2 Conducting depth first search

After input sequences are retrieved, the algorithm conducts, in parallel, B DFSs using these input sequences. To perform DFS, the algorithm uses what we called a *block vector*, which contains a number of *trace sequence vectors* (TSVs), where each TSV is a vector of trace sequences. We now define these terms.

Let us suppose that we have an input sequence $\bar{x}$ and we wish to consider what happens when this is applied in state s of M. There may be multiple possible traces that can occur and a trace sequence will store certain pieces of information about one such trace. Specifically, if the input of $\bar{x}$ in state s can lead to output sequence $\bar{y}$ and state s' then the associated trace sequence will store s, $\bar{y}$, s' and a counter whose values is the length of $\bar{x}$.

Definition 6.4. A trace sequence ν stores a state s, a counter ν_c, a string ω formed by concatenating a state s with an output sequence $\bar{o}$, and a state s'. Thus, $\omega : S.Y^{\star}.S$. We use ν_u to denote the first state, ν_r to denote the output sequence and ν_k to denote the state that is the last element of the string.

As noted, the input of $\bar{x}$ in s might define more than one trace. Further, we might be considering a set S_Δ of states. Let us suppose that the 'current' input sequence (being considered) is $\bar{x}_\Delta$. For each $s \in S_\Delta$, a trace sequence vector will contain the trace sequences defined by $M(s, \bar{x}_\Delta)$. These trace sequences are stored together as a component of a trace sequence vector, along with flag F that is set to True if the algorithm has determined that $\bar{x}_\Delta$ cannot be extended to a $\mathcal{K}$-UIOs for S_Δ.

Definition 6.5. Given input sequence $\bar{x}_\Delta$ and set of states S_Δ, the corresponding trace sequence vector (TSV) Δ is a vector such that each element (χ) of the TSV Δ has a unique state $s \in S_\Delta$, a boolean variable F, and $\mathcal{K}$ trace sequences such that for each trace sequence ν of χ, we have that $\nu_u = s$, $\bar{x}/\nu_r \in L_M(\nu_u)$, and there is a walk from ν_u to ν_k with label $\bar{x}/\nu_r$. Initially F is set to false.

Definition 6.6. A block vector (BV) is a vector of TSVs.

During DFS, the algorithm first receives a multi-set of sets of states $\mathbb{S} = \{S_1, S_2, \ldots, S_B\}$ and B input sequences. Then for each element S_i of $\mathbb{S}$ the algorithm initiates a TSV

	s_1	s_2	s_3	s_4
$w_1 = x_2x_1$	y_1y_0	y_0y_1	y_1y_0	y_1y_1
$w_2 = x_1x_1$	y_1y_0 y_1y_1 y_0y_1	y_1y_1	y_1y_0	y_0y_0 y_1y_0
$w_3 = x_1x_2$	y_1y_1 y_0y_0	y_1y_1	y_1y_1	y_0y_1

Table 2: The traces generated using $B = 3$ for M_2. w_2 is not accepted as it produces more than $\mathcal{K} = 2$ traces.

Δ_i such that $\Delta_i = \{\chi_1, \chi_2, \dots, \chi_n\}$ and hence forms a block vector $\mathcal{B}$ i.e., $\mathcal{B} = \{\Delta_1, \Delta_2, \dots, \Delta_B\}$. Note that initially each TSV Δ has n elements and therefore each BV has $Bn\mathcal{K}$ trace sequences.

Following the formation of the BV, the algorithm calls *DFS kernel* with κ threads where κ is the maximum number of trace sequences in $\mathcal{B}$ ($|\kappa| = Bn\mathcal{K}$). DFS kernel enters a loop that iterates D times. In the jth iteration, t_i attains trace sequence (ν) of the corresponding TSV element (χ) of the corresponding TSV (Δ) from $\mathcal{B}$.

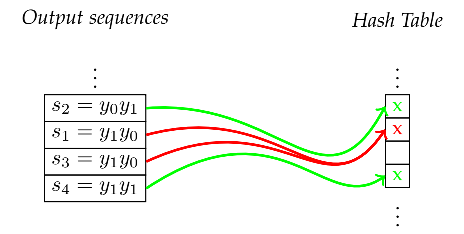
In the DFS kernel, thread t_i receives the current state ν_k, the jth input symbol (x) from the input sequence $\bar{x}_\Delta$ and the FSM transition structure. Then, the thread enters a for loop that iterates $|Y|$ times. At iteration y, the thread checks if there exists a transition leaving ν_k with input x and output y. Let us suppose that such a transition is found. If there exists exactly one transition that leaves ν_k with input x, then the thread updates ν_k with the new current state and appends y to ν_r. Otherwise, t_i selects an empty trace sequence ν' of χ, copies ν_u to ν'_u, ν_k to ν'_k, ν_r to ν'_r, ν_c to ν'_c and replaces the last element of ν'_r with observed output y'. Finally, t_i increments ν_c and ν'_c by one. If $\nu_c > \mathcal{K}$ or x is not defined then the thread sets F to true and the algorithm blocks all threads that process this sequence. Table 1 gives the UPTs computed for M_2.

6.2.3 *Distinguishing states*

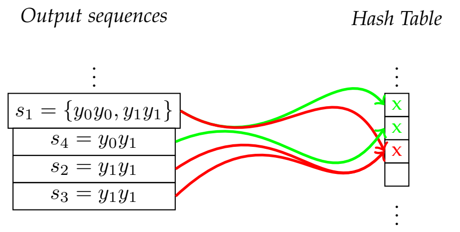
We introduce a new parallel method to check whether a state is distinguished. This method uses sort based hashing to determine whether two states have a common output sequence in response to an input sequence.

First the algorithm extracts the trace sequences from a block vector and sorts output sequences to find whether a state is distinguished using hashing. Let us assume that we are given a block vector $\mathcal{B} = \{\Delta_1, \Delta_2, \Delta_3 \dots \Delta_B\}$. Also let us assume that we have extracted the trace sequences from a TSV Δ_i and form the set $\Psi_{\Delta_i} = \{\bar{\chi}_{r1}, \bar{\chi}_{r2}, \dots, \bar{\chi}_{rn}\}$ where $\bar{\chi}_{rj}$ is the set of output portions of traces observed from a state s_j, i.e., $\bar{\chi}_{rj} = \{\nu_r | \nu \in \Delta_i \wedge \nu_u = s_j\}$ (Line 13 of Algorithm 1). Note that $\bar{\chi}_{ri}$ has at most $\mathcal{K}$ trace sequences.

The algorithm checks whether state $s_j \in S'$ is distinguished as follows. The method first sorts the output portions i.e., `Sort(`Ψ_{Δ_i}`)` (Line 14 of Algorithm 1). Since there are at most $n\mathcal{K}$ traces a sort requires $O(Dn\mathcal{K}log(Dn\mathcal{K}))$ steps. Then for each unique output sequence the algorithm applies a hash function to the output traces and stores them into a hash table according to the hash values (Line 15 of Algorithm 1).



(a) The result using $w_1 = x_2x_1$. The hash based approach indicates that w_2 distinguishes states s_2 and s_4 from any other state.



(b) The result using $w_3 = x_1x_2$. The hash based approach indicates that w_3 distinguishes state s_4 only as one trace from state s_1 and the trace from state s_2 collides in the hash.

Figure 4: Hashing for w_1 and w_3 and FSM M_2.

We use the term *collision* if for a hash table element (or hash element) there is more than one mapping. Thus, s_j is distinguished from all other states of S' if elements in $\bar{\chi}_j$ lead to no collisions. The following is immediate.

Lemma 6.2. Let us suppose that $s_j \in S$ and we have retrieved the output portions of the traces from a block vector Δ_i such that $\bar{\chi}_j \in \Psi_{\Delta_i}$ and there exists no collisions for all traces from state s_j. Then the input sequence x_Δ defines a $\mathcal{K}$-UIOs for s_j.

Lemma Lemma 6.2 leads to the following result.

Theorem 6.1. If there exists a $\mathcal{K}$-UIO of length at most D for s, then the Hi-DFS algorithm finds such a sequence.

For a trace vector Δ, the parallel hash based state distinguishing algorithm executes the above procedure. However, the current implementation does not process all trace vector in $\mathcal{B}$ in parallel[1]. In Figure 4 we demonstrate how hashing allows us to distinguish states of M_2.

state	$\mathcal{K}$-UIOs
s_1	*Not Exists*
s_2	x_2x_1/y_0y_1
s_3	$x_1x_2x_1/y_1y_1y_1$
s_4	x_2x_1/y_1y_1

Table 3: The $\mathcal{K}$-UIOs constructed for M_2.

When the algorithm computes a $\mathcal{K}$-UIO, it checks unique predecessors to derive new $\mathcal{K}$-UIOs in parallel. To achieve this, the algorithm checks the UPT constructed in the first step. For M_2, $w_1 = x_2x_1$ defines $\mathcal{K}$-UIOs for s_2, s_4 as there are no collisions (Figure 4). By using the UPT in Table 1,

1. This might be possible if one has multiple GPUs.

we obtain the $\mathcal{K}$-UIO for state s_3 (through x_1 we reach state s_4). However, the algorithm cannot construct a UIO for state s_1. Note that the unique predecessor for s_1 is $\{s_1, s_2\}$. Although $w_1 = x_2x_1$ is a UIO for s_2 it is not a UIO for s_1. Therefore, because of Definition 3.5, w_1 cannot be used to derive a $\mathcal{K}$-UIO for s_1. Moreover, as D is set to 2 and no input sequence of length 2 can distinguish s_1 from others, the algorithm returns empty set (depicted as $Not\ Exists$) for s_1. The results are given in Table 3.

6.2.4 *Deciding the next step*

The algorithm checks if $\mathcal{K}$-UIOs of some states have been found in the current level, if so the algorithm stores the corresponding input sequences in CPU memory. Afterwards the algorithm decides what to do next. If not all input sequences of length D have been applied and there remain states with no $\mathcal{K}$-UIOs, new input sequences are brought to GPU memory and set $\mathbb{S}$ is formed again and the algorithm continues to search for new $\mathcal{K}$-UIOs. Otherwise, if $\mathcal{K}$-UIOs have been found for all states or the algorithm cannot generate new input sequence(s) then the algorithm terminates.

7 Empirical Study

7.1 Experimental Design

The main motivation of the work described was to explore the notion of $\mathcal{K}$-UIOs for deriving UIOs from ONFSMs. However, due to the absence of a UIO generation algorithm for ONFSMs, we modified the UIO generation algorithm in [33]. We use EA to denote this algorithm[2]. The EA algorithm uses a breadth-first search.

While our primary concern is the ability to construct UIOs, we also recorded the time taken to generate these UIOs and their lengths. The motivation here is that shorter sequences tend to lead to cheaper testing.

We use $N\%$ and $P\%$ to represent the percentage of non-deterministic and partial transitions, respectively. To see the effect of $\mathcal{K}$ branching, we used a set of test suites each containing FSMs generated by the tool used in [43].

Recall that r denotes input alphabet size, v denotes output alphabet size, and n denotes the number of states. In the first test suite (T1), for each $n \in \{10, 20, \ldots, 150\}$ and $\mathcal{K} \in \{4, 8, 16\}$ we constructed 1000 FSMs with $r/v \in \{2/2, 2/4, 2/6, 4/2, 4/6, 6/2, 6/6, 6/6\}$ where $N\% = 10$ and $P\% = 10$. Thus, T1 contained 120,000 FSMs.

We explored the scalability of the Hi-DFS algorithm by running it with larger FSMs. In the second test suite (T2), we fixed the number of inputs/outputs to 6/6 and non-deterministic and partial transitions to $(20, 20)$. For each $n \in \{300, 600, \ldots, 38400\}$ we generated 1,000 FSMs.

There is a threat that the randomly generated FSMs are unlike real FSM specifications. Therefore, we complemented the experiments with case studies: FSM specifications from the ACM/SIGDA benchmarks, a set of FSMs used in workshops in *1989–91–93* [44].

The specifications were in the $kiss2$ format where an input/output is represented by a sequence in $\{0, 1, -\}^*$. Outputs containing $-$ define several transitions. For example, a transition $(st1, 01, 0-, st2)$ defines transitions $(st1, 01, 00, st2)$ and $(st1, 01, 01, st2)$. We found that 32% of the FSMs were observable and non-deterministic. In order to use these FSMs we applied a process that produces the transitions that result from 'completing' a transition with a $-$ symbol. We present properties of these FSMs in Table 4 in which $|h|$ is the number of transitions before the process was applied and $|post(h)|$ is the number of transitions once this process has completed.

2. The modified algorithm is given in http://www.gtu.edu.tr/Files/UIO_OFSM_TR.pdf

7.2 Experiment settings and results

We used an Intel Core I7 CPU (Q6850) with 8GB RAM and NVIDIA TESLA K40 GPU under 64 bit Windows Server 2008 R2. To perform the experiments in an acceptable amount of time, we set 200s as the limiting time.

7.2.1 *The rate of generating* UIOs

We investigated the rate of generating UIOs by comparing the number of times that the algorithms could generate UIOs for FSMs with varying properties and varying $\mathcal{K}$. Let α, β denote the numbers of FSMs for which the Hi-DFS and the EA could construct UIOs, respectively. In Figure 5, we see the averages of the value β/α for T1. For this we used $\mathcal{K} = 4$.

When $n \leq 60$, the EA could generate UIOs regardless of alphabet size. However, when $n \geq 70$ and $r/v = \{4/4, 6/4, 6/6\}$ the EA generated UIOs for at most 85% of the FSMs. When $n = 100$ and $r/v = \{2/2, 4/2, 4/4, 6/2, 6/4\}$ the EA could not generate UIOs for any FSMs. The EA constructed UIOs for 77% of the FSMs when $100 < n \leq 140$ where $r/v = \{2/4, 2/6, 4/6\}$. In contrast, with $\mathcal{K} = 4$, the Hi-DFS algorithm could construct UIOs for all of the FSMs.

7.2.2 *Execution time*

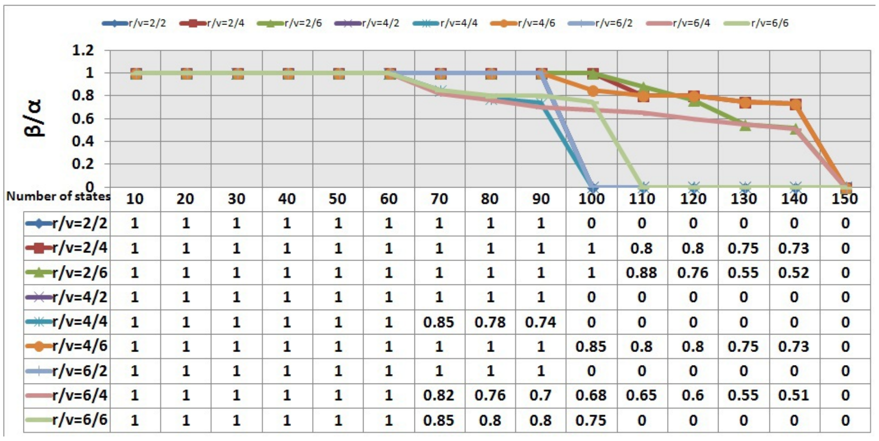
Figure 6 shows the average execution times under varying numbers of transition where $\mathcal{K} \in \{4, 8, 16\}$. While both algorithms require more time as we increase the number of transitions, the rate of increase appears to be higher for the EA algorithm. In addition, the EA algorithm could not generate UIOs when there were more than 800 transitions. The EA algorithm takes much more time than the Hi-DFS algorithm. Figure 7 shows the results for just the Hi-DFS algorithm in order to allow a comparison with different values of $\mathcal{K}$, showing slightly better performance with lower values of $\mathcal{K}$. The overall results have important implications. Although the Hi-DFS algorithm is an exponential algorithm, the time required to compute UIOs grew slowly with the number of transitions. While this may largely stem from the parallel nature of the Hi-DFS algorithm, the computation time is also affected by $\mathcal{K}$.

7.2.3 *Length*

In Figure 8, we see mean (generated) UIO lengths under varying transition sizes and $\mathcal{K} \in \{4, 8, 16\}$. Despite being fast and capable of generating UIOs, the Hi-DFS algorithm generated longer UIOs than the EA algorithm. When $\mathcal{K} = 4$, the Hi-DFS algorithm returns UIOs that are approximately twice the length on average, though the results are better

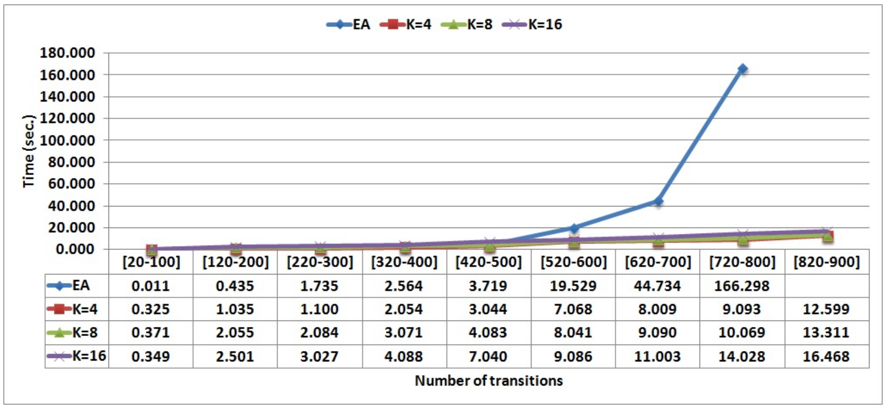
Table 4: Properties of specifications used in the experiments.

Property	$bbse$	cse	$ex2$	$ex3$	$ex5$	$ex7$	$keyb$	$kirkman$	$lion$	$mark1$	$planet$	$sand$	sse	$styr$	$train4$	$train11$
$\|S\|$	16	16	19	10	9	10	19	17	4	16	48	32	16	30	4	11
$\|X\|$	128	128	4	4	4	4	128	4096	4	42	128	2048	128	512	4	4
$\|Y\|$	22	17	4	4	4	4	4	64	2	5132	9292	129	22	211	2	2
$\|h\|$	512	416	13	6	14	13	0	1536	14	0	1664	51072	512	7024	11	19
$\|post(h)\|$	5264	6528	249	126	86	105	10266	228864	16	254656	321648	323712	5264	398256	17	31



Number of states	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150
r/v=2/2	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
r/v=2/4	1	1	1	1	1	1	1	1	1	1	0.8	0.8	0.75	0.73	0
r/v=2/6	1	1	1	1	1	1	1	1	1	1	0.88	0.76	0.55	0.52	0
r/v=4/2	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
r/v=4/4	1	1	1	1	1	1	0.85	0.78	0.74	0	0	0	0	0	0
r/v=4/6	1	1	1	1	1	1	1	1	1	0.85	0.8	0.8	0.75	0.73	0
r/v=6/2	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
r/v=6/4	1	1	1	1	1	1	0.82	0.76	0.7	0.68	0.65	0.6	0.55	0.51	0
r/v=6/6	1	1	1	1	1	1	0.85	0.8	0.8	0.75	0	0	0	0	0

Figure 5: Average rate of generating $\mathcal{K}$-UIOs for the FSMs in T1.

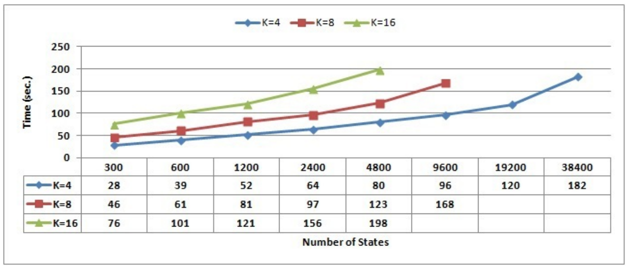


	[20-100]	[120-200]	[220-300]	[320-400]	[420-500]	[520-600]	[620-700]	[720-800]	[820-900]
EA	0.011	0.435	1.735	2.564	3.719	19.529	44.734	166.298	
K=4	0.325	1.035	1.100	2.054	3.044	7.068	8.009	9.093	12.599
K=8	0.371	2.055	2.084	3.071	4.083	8.041	9.090	10.069	13.311
K=16	0.349	2.501	3.027	4.088	7.040	9.086	11.003	14.028	16.468

Figure 6: Mean execution time for FSMs in test suite T1.

when $\mathcal{K} > 4$. The results are as expected since the Hi-DFS algorithm uses a depth-first search; the breadth-first EA algorithm algorithm can find shorter UIOs.

7.3 Scalability



	300	600	1200	2400	4800	9600	19200	38400
K=4	28	39	52	64	80	96	120	182
K=8	46	61	81	97	123	168		
K=16	76	101	121	156	198			

Figure 9: $\mathcal{K}$-UIO generation time for FSMs in test suite T2.

Recall that T2 was used to explore scalability as the number of states increases. Figure 9 provides $\mathcal{K}$-UIO generation time for T2. For $\mathcal{K} = 16$, the Hi-DFS algorithm could construct UIOs for $n < 9600$. The maximum n becomes 19,200 ($\mathcal{K} = 8$) and 38,400 ($k = 4$). The result are promising; the Hi-DFS algorithm is 250 times more scalable than the EA algorithm.

7.4 Benchmark FSMs

We applied the Hi-DFS algorithm and EA to each FSM in the benchmark. Figure 10 presents the percentages of states for which the algorithms constructed UIOs. EA was able to generate UIOs for at most 2 states (12.5%) of FSMs bbse, cse, sand, sse and styr but constructed UIOs for the remaining FSMs. So, the EA was capable of constructing UIOs for 56.25% of the FSMs.

On the other hand, when $\mathcal{K} = 8$, the $\mathcal{K}$-UIO algorithm could construct UIOs for every FSM in the benchmark test suite. However when $\mathcal{K} = 4$, the $\mathcal{K}$-UIO algorithm could not generate UIOs for 1 and 2 states of FSMs Mark1 and Planet respectively as both specifications have $\mathcal{K}$-UIO with at least 5 traces, i.e., $\mathcal{K} > 4$.

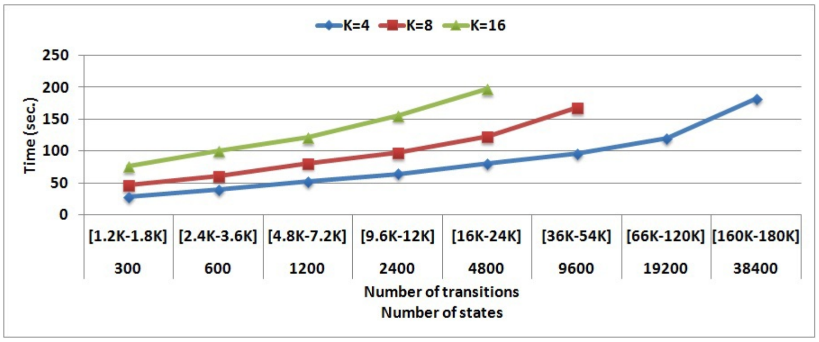
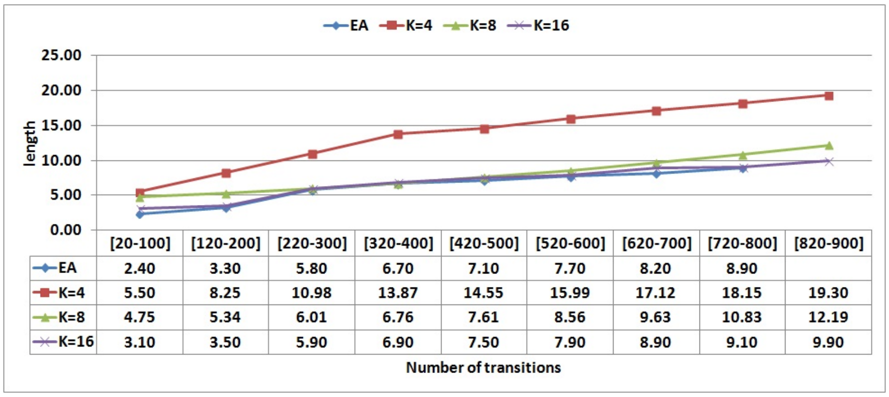


Figure 7: Mean execution time for Hi-DFS under varying transition sizes.



	[20-100]	[120-200]	[220-300]	[320-400]	[420-500]	[520-600]	[620-700]	[720-800]	[820-900]
EA	2.40	3.30	5.80	6.70	7.10	7.70	8.20	8.90	
K=4	5.50	8.25	10.98	13.87	14.55	15.99	17.12	18.15	19.30
K=8	4.75	5.34	6.01	6.76	7.61	8.56	9.63	10.83	12.19
K=16	3.10	3.50	5.90	6.90	7.50	7.90	8.90	9.10	9.90

Figure 8: Mean $\mathcal{K}$-UIO length under varying transition sizes.

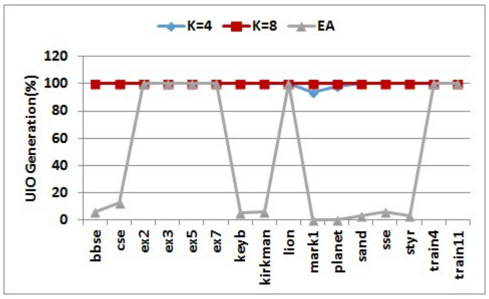


Figure 10: Percentage of states with generated $\mathcal{K}$-UIOs (benchmarks).

Figure 11 gives UIO generation time using a log scale (base 10). The EA algorithm could not finish computation in 200s for bbse, cse, sand, sse and styr. As expected, when $\mathcal{K}$ increased to 8, UIO construction time increased.

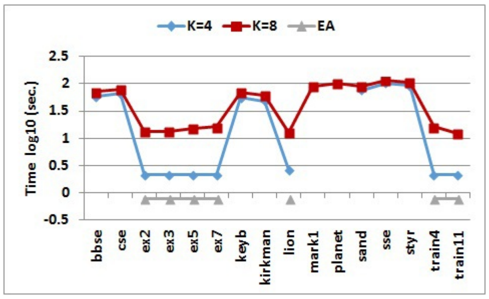


Figure 11: $\mathcal{K}$-UIO generation time (benchmarks).

Average UIO lengths are given in Figure 12, with these not exceeding 6 inputs. We observed that the EA algorithm generated shorter UIOs than the Hi-DFS algorithm. The result is similar to that found with randomly generated FSMs and we believe that this stems from the EA algorithm using breadth-first search.

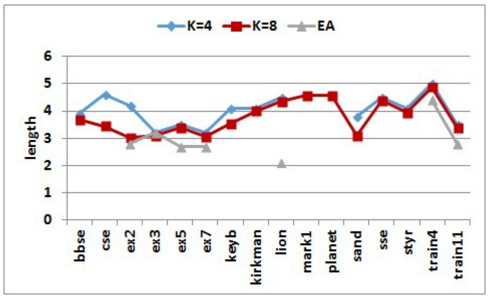


Figure 12: Mean $\mathcal{K}$-UIO length (benchmarks).

It has been reported that only one of the benchmark FSMs (sand) has a distinguishing sequence [43]; distinguishing sequence based test generation techniques cannot be used for the other benchmark FSMs. In contrast, we were able to generate $\mathcal{K}$-UIOs and so $\mathcal{K}$-UIOs could form the basis for test generation for these real FSMs.

7.5 Discussion

Despite the importance of non-determinism, there are not many algorithms that can generate state identification sequences for ONFSMs. A crucial difficulty lies in the fact that the number of traces generated from the FSM grows exponentially, with this clearly affecting scalability. In this paper, we considered UIOS as state identification sequences, the concept of $\mathcal{K}$-UIOS allowing us to directly address scalability. Importantly, in the experiments we found that we could construct UIOS for randomly generated and real FSMs, including FSMs for which the EA fails. In particular, the proposed massively parallel algorithm was able to construct UIOS for (real or randomly generated) FSMs with 38K states and 400K transitions.

7.6 Threats to validity

Naturally, there are several potential threats to validity. We start by discussion threats to generalisability, which relate to the ability to generalise results.

First, consider the randomly generated FSMs. Different FSMs might lead to different results but we reduced this threat by generating many FSMs. The FSM (random) generator has a number of parameters (e.g. number of states) and it is possible that different values might have led to different results. In order to (partially) address this we used a number of different values for the parameters.

A danger with using randomly generated FSMs is that 'real' FSMs might be rather different. To reduce this threat we used benchmark FSMs and found that the results were similar to those produced using randomly generated FSMs. However, there would be value in using additional FSMs from a range of application domains.

Another potential threat is that we might have incorrectly implemented the algorithms. To address this we carefully tested the code. We also used an existing tool [43] to check all UIOS generated.

8 Conclusions

In model based testing, test cases are derived from a model and then applied to the SUT. Much of the focus has been on testing from DFSMs but there is an increasing need to test from observable non-deterministic finite state machines (ONFSMs). Many algorithms that generate tests from DFMSs use UIOS and in this paper we extended the notion of UIOS to ONFSMs and explored the UIO generation problem for ONFSMs. Non-determinism introduces a scalability challenge: there may be exponentially many specified responses to an input sequence. We addressed this engineering problem by introducing $\mathcal{K}$ branching UIOS: UIOS that can lead to at most $\mathcal{K}$ different observations. We proved that checking $\mathcal{K}$-UIOS existence is PSPACE Complete even if the UIO length is bounded by a polynomial in the number of states of the ONFSM. We introduced a massively parallel algorithm for generating $\mathcal{K}$-UIOS, with experiments showing that this scaled much better than a classical UIO generation algorithm.

The experiments indicated that the notion of $\mathcal{K}$-branching helps us make UIO generation more scalable. One particular line of future work is to extend this to other important sequences used in test generation such as adaptive distinguishing sequences, preset distinguishing sequences [43] characterizing sets and harmonized state identifiers [45]. There may also be scope to use SAT solvers or constraint solvers to generate $\mathcal{K}$-UIOS, potentially adapting work that generates UIOs [46]. Finally, there would also be value in additional experiments with FSMs from industry, ideally using more recent examples. One possible source of such FSMs was recently identified by researchers who used FSMs based on intrusion detection [47].

Acknowledgements

This work is partially supported by AUS under grant eFRG18-SET-CEN-49.

References

[1] A. Friedman and P. Menon, *Fault detection in digital circuits*, ser. Computer Applications in Electrical Engineering Series. Prentice-Hall, 1971.

[2] A. Aho, R. Sethi, and J. Ullman, *Compilers, principles, techniques, and tools*, ser. Addison-Wesley series in computer science. Addison-Wesley Pub. Co., 1986.

[3] T. S. Chow, "Testing software design modelled by finite state machines," *IEEE Transactions on Software Engineering*, vol. 4, pp. 178–187, 1978.

[4] D. Lee, K. Sabnani, D. Kristol, and S. Paul, "Conformance testing of protocols specified as communicating finite state machines-a guided random walk based approach," *IEEE Transactions on Communications*, vol. 44, no. 5, pp. 631–640, May.

[5] D. Lee and M. Yannakakis, "Principles and methods of testing finite-state machines - a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1089–1123, 1996.

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

[7] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[8] 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*, ser. LNCS, vol. 3235. Springer-Verlag, 2004, pp. 115–132.

[9] A. Betin-Can and T. Bultan, "Verifiable concurrent programming using concurrency controllers," in *Proceedings of the 19th IEEE international conference on Automated software engineering*. IEEE Computer Society, 2004, pp. 248–257.

[10] I. Pomeranz and S. M. Reddy, "Test generation for multiple state-table faults in finite-state machines," *IEEE Transactions on Computers*, vol. 46, no. 7, pp. 783–794, 1997.

[11] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[12] 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*, vol. 21, no. 1, pp. 55–71, 2011.

[13] 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," in *Protocol Specification, Testing, and Verification VIII*. Atlantic City: Elsevier (North-Holland), 1988, pp. 75–86.

[14] F. C. Hennie, "Fault-detecting experiments for sequential circuits," in *Proceedings of Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, November 1964, pp. 95–110.

[15] G. Gonenc, "A method for the design of fault detection experiments," *IEEE Transactions on Computers*, vol. 19, pp. 551–558, 1970.

[16] S. T. Vuong, W. Y. L. Chan, and M. R. Ito, "The UIOv-method for protocol test sequence generation," in *The 2nd International Workshop on Protocol Test Systems*, Berlin, 1989.

[17] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test selection based on finite state models," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 591–603, 1991.

[18] K. El-Fakih, N. Yevtushenko, and G. v. Bochmann, "FSM-based incremental conformance testing methods," *IEEE Transactions on Software Engineering*, vol. 30, no. 7, pp. 425–436, 2004.

[19] A. Petrenko and N. Yevtushenko, "Testing from partial deterministic FSM specifications," *IEEE Transactions on Computers*, vol. 54, no. 9, pp. 1154–1165, 2005.

[20] R. Dorofeeva, K. El-Fakih, S. Maag, A. R. Cavalli, and N. Yevtushenko, "FSM-based conformance testing methods: a survey annotated with experimental evaluation," *Information and Software Technology*, vol. 52, no. 12, pp. 1286–1297, 2010.

[21] A. T. Endo and A. da Silva Simão, "Evaluating test suite characteristics, cost, and effectiveness of FSM-based testing methods," *Information & Software Technology*, vol. 55, no. 6, pp. 1045–1062, 2013.

[22] A. Aho, A. Dahbura, D. Lee, and M. Uyar, "An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours," *IEEE Transactions on Communications*, vol. 39, no. 11, pp. 1604 –1615, nov 1991.

[23] W. Y. L. Chan, C. T. Vuong, and M. R. Otp, "An improved protocol test generation procedure based on UIOs," *SIGCOMM Computer Communication Review*, vol. 19, no. 4, pp. 283–294, Aug. 1989.

[24] W.-H. Chen and H. Ural, "Synchronizable test sequences based on multiple UIO sequence," *IEEE/ACM Transactions on Networking*, vol. 3, no. 2, pp. 152–157, 1995.

[25] S. Guyot and H. Ural, "Synchronizable checking sequences based on UIO sequences," in *Protocol Test Systems, VIII*. Evry, France: Chapman and Hall, September 1995, pp. 385–397.

[26] H. Motteler, A. Chung, and D. Sidhu, "Fault coverage of UIO-based methods for protocol testing," in *Proceedings of Protocol Test Systems VI*, 1994, pp. 21–33.

[27] T. Ramalingam, K. Thulasiraman, and A. Das, "A generalization of the multiple UIO method of test sequence selection for protocols represented in FSM," in *The 7th International workshop on Protocol Test Systems*, 1994, pp. 209–224.

[28] H. Ural and Z. Wang, "Synchronizable test sequence generation using UIO sequences," *Computer Communications*, vol. 16, no. 10, pp. 653–661, 1993.

[29] D. Lee and M. Yannakakis, "Testing finite-state machines: State identification and verification," *IEEE Transactions on Computers*, vol. 43, no. 3, pp. 306–320, 1994.

[30] I. Ahmad, F. Ali, and A. Das, "LANG-algorithm for constructing unique input/output sequences in finite-state machines," in *Computers and Digital Techniques, IEE Proceedings-*, vol. 151, no. 2. IET, 2004, pp. 131–140.

[31] Q. Guo, R. M. Hierons, M. Harman, and K. Derderian, "Computing unique input/output sequences using genetic algorithms," in *Formal Approaches to Software Testing*. Springer, 2004, pp. 164–177.

[32] ——, "Constructing multiple unique input/output sequences using metaheuristic optimisation techniques," *IEE Proceedings-Software*, vol. 152, no. 3, pp. 127–140, 2005.

[33] K. Naik, "Efficient computation of unique input/output sequences in finite-state machines," *IEEE/ACM Transactions on Networking*, vol. 5, no. 4, pp. 585–599, Aug. 1997.

[34] R. M. Hierons and U. C. Türker, "Parallel algorithms for testing finite state machines: Generating UIO sequences." *IEEE Transactions on Software Engineering*, vol. 42, no. 11, pp. 1077–1091, 2016.

[35] Z. Kohavi, *Switching and Finite State Automata Theory*. McGraw-Hill, New York, 1978.

[36] P. H. Starke, *Abstract Automata*. Elsevier, North-Holland, Amsterdam, 1972.

[37] K. El-Fakih, R. Dorofeeva, N. Yevtushenko, and G. von Bochmann, "FSM-based testing from user defined faults adapted to incremental and mutation testing," *Programming and Computer Software*, vol. 38, no. 4, pp. 201–209, 2012.

[38] R. Alur, C. Courcoubetis, and M. Yannakakis, "Distinguishing tests for nondeterministic and probabilistic machines," in *27th ACM Symposium on Theory of Computing*, 1995, pp. 363–372.

[39] N. Spitsyna, K. El-Fakih, and N.Yevtushenko, "Studying the separability relation between finite state machines," *Software Testing, Verification and Reliability*, vol. 17, no. 4, pp. 227–241, 2007.

[40] N. Kushik, K. El-Fakih, and N. Yevtushenko, "Adaptive homing and distinguishing experiments for nondeterministic finite state machines," in *Testing Software and Systems*, ser. Lecture Notes in Computer Science, H. Yenigün, C. Yilmaz, and A. Ulrich, Eds., vol. 8254. Springer Berlin Heidelberg, 2013, pp. 33–48.

[41] N. Kushik, K. El-Fakih, N. Yevtushenko, and A. R. Cavalli, "On adaptive experiments for nondeterministic finite state machines," *International Journal of Software Tools for Technology Transfer*, vol. 18, no. 3, pp. 251–264, 2016.

[42] W. J. Savitch, "Relationships between nondeterministic and deterministic tape complexities," *Journal of Computer and System Sciences*, vol. 4, no. 2, pp. 177 – 192, 1970.

[43] R. M. Hierons and U. C. Türker, "Parallel algorithms for generating distinguishing sequences for observable non-deterministic fsms," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 1, pp. 5:1–5:34, 2017.

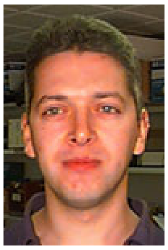
[44] F. Brglez, "ACM/SIGMOD benchmark dataset," Available online at http://www.cbl.ncsu.edu/benchmarks/Benchmarks-upto-1996.html, 1996, accessed: 2014-02-13.

[45] R. M. Hierons and U. C. Türker, "Parallel algorithms for testing finite state machines: Harmonised state identifiers and characterising sets," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3370–3383, 2016.

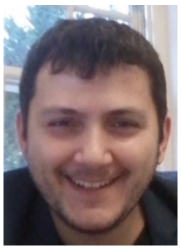
[46] C. Güniçen, U. C. Türker, H. Ural, and H. Yenigün, "Generating preset distinguishing sequences using SAT," in *Computer and Information Sciences II*, E. Gelenbe, R. Lent, and G. Sakellari, Eds. Springer London, 2012, pp. 487–493, 10.1007/978-1-4471-2155-8_62. [Online]. Available: http://dx.doi.org/10.1007/978-1-4471-2155-8_62

[47] V. Yaneva, A. Kapoor, A. Rajan, and C. Dubach, "Accelerated finite state machine test execution using GPUs," in *25th Asia-Pacific Software Engineering Conference (APSEC 2018)*, 2018.

Khaled El-Fakih received BS and MS degrees in Computer Science from the Lebanese American University and a Ph.D. in Computer Science from the University of Ottawa. He is full professor at the Department of Computer Science and Engineering at the American University of Sharjah. His research work includes formal testing, automatic synthesis of distributed systems, optimisation and application of genetic algorithms.



Robert M Hierons received a BA in Mathematics (Trinity College, Cambridge), and a Ph.D. in Computer Science (Brunel University). He then joined the Department of Mathematical and Computing Sciences at Goldsmiths College, University of London, before returning to Brunel University in 2000. He was promoted to full Professor in 2003 and joined The University of Sheffield In 2018.



Uraz Cengiz Türker is an assistant professor of Computer Engineering at Gebze Technical University. He received the BA, MSc and PhD degrees in Computer Science (Sabanci University, Turkey), in 2006, 2008, and 2014, respectively. He worked with Prof. Robert M. Hierons as a post doctorate researcher at Brunel University London.