Parallel Algorithms for Testing Finite State Machines: Generating UIO sequences

Robert M. Hierons, *Senior Member, IEEE* and Uraz Cengiz Türker

Abstract—This paper describes an efficient parallel algorithm that uses many-core GPUs for automatically deriving Unique Input Output sequences (UIOs) from Finite State Machines. The proposed algorithm uses the global scope of the GPU's global memory through coalesced memory access and minimises the transfer between CPU and GPU memory. The results of experiments indicate that the proposed method yields considerably better results compared to a single core UIO construction algorithm. Our algorithm is scalable and when multiple GPUs are added into the system the approach can handle FSMs whose size is larger than the memory available on a single GPU.

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

Software testing is an important part of the software development process but is typically expensive, manual and error prone. This has led to significant interest in automation and one of the most promising approaches is *model-based testing* (MBT) in which test automation is based on a model of the *system under test* (SUT) or some aspect of the SUT. Many MBT methods base test automation on a *finite state machine* (FSM), with this line of work going back to Moore's seminal paper [1].

Many FSM-based test generation methods check that the transitions of the FSM specification M have been implemented correctly. In order to check a transition it is necessary to have some method that checks that the state of the SUT, after input x in state s, is the expected state s'. This is typically achieved by using input sequences that distinguish the states of M. Ideally, one has a distinguishing sequence (an input sequence that distinguishes all of the states of M) and early work by Hennie showed how a test sequence can be automatically derived when there is a known distinguishing sequence [2]. However, an FSM need 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 but 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 such UIOs [3] and this has led to the development of many FSM-based test generation methods that use UIOs [3], [4], [5], [6], [7], [8], [9], [10], [11]. However, it is known that the problem of checking the existence of a UIO is PSPACE-Hard and so one cannot expect to find polynomial time algorithms that construct UIOs and there is no polynomial upper bound on UIO length. Since the length of a UIO sequence can be exponential, the duration and hence the cost of deriving such sequences from large FSMs can be very high. This has led to interest in methods that relatively efficiently generate UIOs [12], [13], [14], [15] but ultimately we cannot get away from the worst case complexity. Previous approaches for deriving UIOs have developed sequential algorithms that operate on a single thread and have not used *Graphics Processing Units (GPUs)* despite the increasing interest in GPUs. Recently, with the publication of the *Compute Unified Device Architecture (CUDA)* development toolkit that allows GPU programming in a C-like language, the use of GPUs has been extended to a range of application domains [16], [17], [18], [19], [20].

In this paper, we address the scalability problem that can arise while constructing UIOs for large completely-specified FSMs through the use of massively parallel GPU technology. The work was motivated by the fact that GPUs have become an important tool in large scale applications in which massively parallel processing is needed. As far as we are aware, this problem has not previously been explored. One of the reasons for this may be that there is a need to model the UIO generation problem in a manner that is suitable for GPUs and this is not straightforward; previous algorithms for constructing UIOs have used data structures that are not suitable for GPU computing. As previously noted [21] this can be considered to be the biggest difference between CPUs and GPUs: much of the physical space of a GPU is reserved for computing units rather than memory. This space is divided into many relatively simple cores, with the instruction set of a core being much smaller than that

• *Department of Computer Science, Brunel University London, UK. E-mail: {rob.hierons,uraz.turker}@brunel.ac.uk*

of a standard CPU. Despite their limited instruction sets, the performance of GPUs make them highly effective when used to solve certain types of problems. In addition, the demand for high performance graphics (due to computer gaming, high resolution image processing and big data research) led to increasing parallelism. In fact, new massively parallel computing processors such as the NVIDIA's Tesla-K40 already have 2880 cores where a core has clock speed of 745 MHz, 288 GB/sec memory bandwidth, and 12 GB of memory.

The constraints imposed by GPU computing led to us devising an algorithm that is entirely new with the exception of the fact that it utilises the 'Unique Predecessor' approach devised by Naik [15]. Otherwise the proposed algorithm is unique since all the existing brute force approaches in the FSM based literature construct a 'UIO Tree' and this is not appropriate due to the space/time limitations (as shown by the experiments). There were several additional challenges related to memory management. These challenges include the need to efficiently distribute data processing between the CPU and GPU, thread synchronisation, optimisation of data transfer, and the capacity constraints of GPU memory.

This paper proposes a massively parallel UIO (P-UIO) generation algorithm that addresses these problems in the context of deriving UIOs for a deterministic completely specified FSM. The P-UIO algorithm was evaluated against Naik's algorithm using randomly generated FSMs with up to 1,048,576 states. In the experiments the proposed algorithm constructed UIOs significantly faster (by a factor of 11000 on average) and for much larger FSMs (by a factor of 512). For example, the P-UIO algorithm was able to handle FSMs with 1,048,576 states in under 2 seconds on average while the implementation of Naik's algorithm took 1231 seconds on average for FSMs with 2048 states. We also performed experiments on some much smaller benchmark FSMs, with between 4 and 48 states. The performance of the two algorithms was similar for these FSMs but the P-UIO algorithm took less time to find UIOs for the larger benchmark FSMs. Unsurprisingly, the differences in performance were less significant for the (much smaller) benchmark FSMs.

This paper is organised as follows. Section 2 introduces the terminology used and reviews previously devised UIO generation techniques. Section 3 provides an overview of the proposed parallel UIO algorithm and describes the high-level design. Section 4 provides the low-level design. In Section 5 we describe the experiments designed to evaluate the proposed UIO construction algorithm and the results of these experiments. Finally, in Section 6 we provide concluding remarks and discuss possible lines of future work.

2 PRELIMINARIES

2.1 Finite State Machines (FSMs)

A finite state machine (FSM) M is defined by a tuple $(S, X, Y, \delta, \lambda)$ where S is a finite set of states, $X = \{x_1, x_2, \ldots, x_p\}$ is a finite set of inputs, $Y = \{o_1, o_2, \ldots, o_r\}$ is a finite set of outputs, δ is the transition function of type $\delta : S \times X \rightarrow S$ and λ is the output function of type $\lambda : S \times X \rightarrow Y$. The functions δ and λ are total functions and so M is *completely-specified*. If FSM M is in state $s \in S$ and input $x \in X$ is applied then M moves to the state $s' = \delta(s, x)$ and produces output $o = \lambda(s, x)$. Such a *transition* will be denoted $\tau = (s, x/o, s')$ and we say that x/o is the *label* of τ ($label(\tau)$), s is the *start state* of τ ($start(\tau)$), and s' is the *end state* of τ ($end(\tau)$). Note that sometimes the definition of an FSM includes an initial state; we do not include this since we are interested in distinguishing states of an FSM and so do not require there to be an initial state.

We use juxtaposition to denote concatenation: if x_1, x_2, and x_3 are inputs then $x_1x_2x_3$ is an input sequence. Given a set X we let X^* denote the set of finite sequences of elements of X and let X^k denote the set of sequences in X^* that have length k. The symbol ε is used to denote the empty sequence.

An input/output sequence consists of a sequence of input/output pairs of the form $x_1/o_1\ x_2/o_2 \ldots x_m/o_m$. We will also write $x_1x_2 \ldots x_m/o_1o_2 \ldots o_m$ to denote this input/output sequence, where $x_1x_2 \ldots x_m$ is called the *input portion* and $o_1o_2 \ldots o_m$ is called the *output portion* of the input/output sequence. A *path* in M is a sequence of transitions $\bar{\tau} = \tau_1\tau_2 \ldots \tau_m$ such that $start(\tau_i) = end(\tau_{i-1})$, for all $1 < i \leq m$. The *label* of a path is an input/output sequence which is the concatenation of the labels (input/output pairs) of the transitions in that path. For path $\bar{\tau} = \tau_1\tau_2 \ldots \tau_m$, we define $label(\bar{\tau}) = label(\tau_1)label(\tau_2) \ldots label(\tau_m)$.

The transition and output functions can be extended to input sequences as follows. For $\bar{x} \in X^\star$ and $x \in X$, $\bar{\delta}(s, \varepsilon) = s$, $\bar{\delta}(s, x\bar{x}) = \bar{\delta}(\delta(s, x), \bar{x})$, $\bar{\lambda}(s, \varepsilon) = \varepsilon$, $\bar{\lambda}(s, x\bar{x}) = \lambda(s, x)\bar{\lambda}(\delta(s, x), \bar{x})$. We call a subset $B \subseteq S$ of states a *group*. The transition and output functions are extended to groups as follows. For a group B and $\bar{x} \in X^\star$, $\bar{\delta}(B, \bar{x}) = \cup_{s \in B}\{\bar{\delta}(s, \bar{x})\}$ and $\bar{\lambda}(B, \bar{x}) = \cup_{s \in B}\{\bar{\lambda}(s, \bar{x})\}$. We will use δ and λ to denote $\bar{\delta}$ and $\bar{\lambda}$, respectively.

An input sequence $\bar{x} \in X^\star$ is a *splitting sequence* for a group B, if $|\lambda(B, \bar{x})| > 1$. Such an input sequence $\bar{x}$ leads to different output sequences from at least two states in B and so $\bar{x}$ *splits* B. We call an input x a *splitting input* for B if x is a splitting sequence of length one for B. If for every pair of states of FSM M there exists a splitting sequence then M is said to be *minimal*.

In this work, we consider only deterministic, completely-specified, minimal FSMs. An FSM can be minimised in polynomial time [22]. Further, an FSM that is not completely-specified can often be completed by adding either an error state or transitions with null output[1]. Thus, the main restriction is that we only consider deterministic FSMs. While non-determinism can be a useful abstraction technique, and some classes of

1. As has been previously noted, it is not always possible to complete an FSM since, for example, unspecified input may correspond to input that should not occur [23].

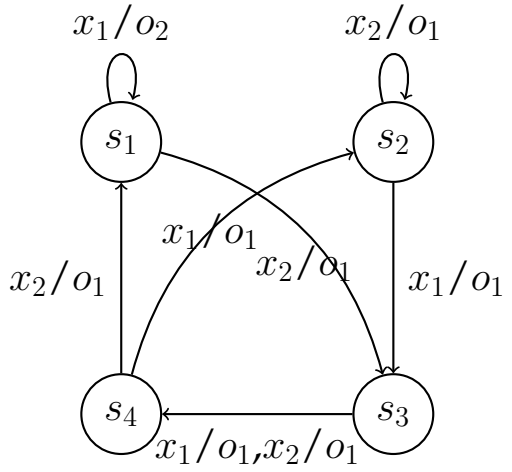


Figure 1: An example deterministic, completely specified and minimal FSM M_1.

systems are non-deterministic, the main focus of FSM-based testing work has been on deterministic FSMs and these have been found to be sufficient in important application domains such as hardware [24], protocol conformance testing [5], [25], [26], [27], [28], [29], object-oriented systems [30], web services [31], [32], [33], [34], and general software [35].

An FSM M can be represented by using a directed graph G where the vertices of G correspond to the states of M and the edges of G correspond to the transitions of M. An FSM M is *strongly connected* if the corresponding directed graph G is strongly connected (for any ordered pair (v, v') of vertices there is a path from v to v'). In Figure 1 an example FSM M_1 is given, where $S = \{s_1, s_2, s_3, s_4\}$, $X = \{x_1, x_2\}$, and $Y = \{o_1, o_2\}$. It is straightforward to see that this FSM is strongly connected. Note that M_1 is a minimal machine, since the input sequence $x_1x_2x_1x_1x_2x_1$ is a splitting sequence for every pair of different states.

For a given FSM, state verification can be achieved by using a *Distinguishing sequence (DS)*, *unique input/output sequences (UIOs)*, a *Characterising set (CS)*, or *state identifiers*. A distinguishing sequence is an input sequence $\bar{x}$ such that for every pair (s, s') of distinct states of FSM M, M produces different output sequences in response to $\bar{x}$ from s and s' ($\lambda(s, \bar{x}) \neq \lambda(s', \bar{x})$). Unfortunately, not all FSMs have a DS. An alternative is to use an *adaptive distinguishing sequence*, which is an adaptive process that determines the next input to apply on the basis of the output observed. Since ADSs generalise DSs, an FSM that has a DS also has an ADS and an additional benefit is that it is possible to determine whether an FSM has an ADS in low-order polynomial time [36].

A UIO for state s is an input/output sequence $\bar{x}/\bar{o}$ such that $\lambda(s, \bar{x}) = \bar{o}$ and for all $s' \in S \setminus \{s\}$, we have that $\lambda(s', \bar{x}) \neq \bar{o}$. While not every FSM has UIOs for all states, some FSMs without a DS or ADS have UIOs for all states. There may be value in using UIOs even when an FSM has a DS since the UIOs may be shorter than the DS. For example, state s_1 of M_1 has a UIO (x_1/o_2) of length 1 but it is straightforward to check that M_1 does not have a DS of length 1. It has also been found that in practice many FSMs have UIOs for all states [3].

A characterising set (CS) is a set W of input sequences that can distinguish any pair of states. A minimal FSM with n states has a CS with at most $n-1$ sequences of length at most $n-1$ and such a CS can be found in polynomial time. If every sequence in W is executed from state s, the set of output sequences identifies/verifies s. However, the use of characterising sets could lead to long test sequences [37]. In addition, most test generation techniques that use a characterising set return many test sequences and it has been noted that the process of resetting a system between test sequences can be expensive [38], [39], [40], [41]. As a result, it is desirable to use a DS or UIOs where they exist (and are sufficiently short) and use a characterising set otherwise. Since the size of a characterising set is of $O(n^2)$, it has been suggested that if a UIO or DS is longer than an $O(n^2)$ upper bound then it might be best to use a characterising set. This has led to the suggestion that one might initially attempt to find a DS or UIOs but only use a DS/UIOs if the length is lower that the given bound. There has been much interest in UIOs [42], [43] since they help in state transition fault detection and have been found to yield shorter test sequences than using the DSs and CSs [42], [43].

It has been observed that it may not be necessary to use all of the sequences from a characterising set in order to identify a state s of the FSM [44]. This has led to the notion of state identifiers, where a state identifier (or separating set) for state s is a set of input sequences that distinguish s from all other states of the FSM M. The use of state identifiers can lead to smaller test suites, when compared to characterising sets.

2.2 Previous UIO generation methods and Inference Rules

Since UIOs have been used in automated FSM-based test generation methods, there has been interest in the problem of devising UIOs. Although the problem of checking the existence of UIOs is PSPACE-Complete [36], the value of UIOs in test generation has led to significant interest in UIO generation.

It is possible to represent UIO generation in terms of a UIO tree (Definition 2.1) [15].

Definition 2.1: Let M be an FSM with set of states S ($|S| = n$). A *Unique Input Output Tree* for S is a rooted tree T such that nodes are labeled with two groups (initial group I and current group C) and edges are labeled with input/output pairs. A node of T is called a *leaf node* if its groups have cardinality one. If two distinct edges leaving a node v share the same input label then these edges have different output labels. For every node v of T, if $\bar{x}/\bar{o}$ is the input/output sequence formed by

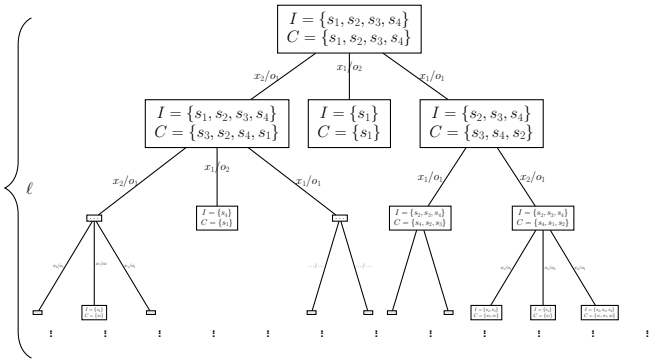


Figure 2: An example of a UIO Tree of depth ℓ for FSM M_1 given in Figure 1.

concatenating the edge labels on the path from the root node to v, then we have that $I = \{s \in S | \lambda(s, \bar{x}) = \bar{o}\}$ and $C = \cup_{s \in I}\{\delta(s, \bar{x})\}$. A UIO tree is *complete* if for every state s_i there exists a leaf node v with initial set $I = \{s_i\}$.

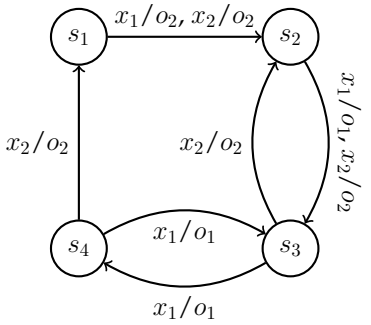
An example UIO tree for M_1 is given in Figure 2.

As the upper bound on UIO length is exponential, the process of constructing UIO trees from scratch can be expensive. This led to Naik [15] proposing an approach to construct UIOs in which *inference rules* are used. In this method some minimal length UIOs are found and these UIOs are used to deduce UIOs for other states. The inference rules operate as follows: If $\bar{x}/\bar{o}$ is a UIO for state s and $\tau = (s', x/o, s)$ is the only transition that reaches state s with label x/o, then $x\bar{x}/o\bar{o}$ is a UIO for s'. Here state s' is known as a *unique predecessor* of state s. Note that the notion of 'unique' here is for the state and input/output; there might be more than one unique predecessor for a state s. Since a machine M is assumed to be strongly connected, it may be possible to use inference rules to compute UIO sequences for all states once we have constructed only a few UIOs. In order to achieve this we have a database, called a *rule base*, containing the known inference rules for the FSM. In the following, we formalise what it means for a state to be a unique predecessor.

Definition 2.2: For state $s \in S$, $s' \in S$ is a *unique predecessor* for s if there exists a transition τ with $start(\tau) = s'$, $end(\tau) = s$ and $label(\tau) = x/o$ such that there exists no transition $\tau' \neq \tau$ with $end(\tau) = s$ and $label(\tau) = x/o$.

As an example, see the FSM M_2 in Figure 3a. Note that for state s_1 the unique predecessor is s_4 with transition labeled by x_2/o_2, for state s_4 the unique predecessor is s_3 with transition labeled by x_1/o_1, for state s_3 the unique predecessor is s_2 with transition labeled by x_2/o_2 and finally for state s_2 the unique predecessor is s_1 with transition labeled by x_1/o_2. The rule base table for FSM M_2 is given in Figure 3b. Let us assume that by using a UIO tree we can construct a UIO for state s_1. Then by using the rule base table, we can derive UIOs for all of the states of the FSM (Figure 3c).

If an FSM does not possess UIOs for all of its states, the inference rules do not solve the problem. Naik's algorithm then constructs the full UIO tree, which requires exponential time/space.



(a) A deterministic, completely specified and minimal FSM M_2

State	*unique predecessor*	τ
s_1	s_4	x_2/o_2
s_2	s_1	x_1/o_2
s_3	s_2	x_2/o_2
s_4	s_3	x_1/o_1

(b) Rule base (rb) table for FSM M_2 given in Figure 3a

State	UIO
s_1	$\mathbf{x_1/o_2}$
s_2	$x_2x_1x_2x_1/o_2o_1o_2o_2$
s_3	$x_1x_2x_1/o_1o_2o_2$
s_4	x_2x_1/o_2o_2

(c) Assume that UIO for state s_1 is computed throught a UIO tree (bold typed). Then by using the rule base table given in Figure 3b we can derive UIOs for other states.

Figure 3: An FSM M_2 with its rule base. After the UIO for s_1 is computed, UIOs for every other state can be obtained by using the rule base.

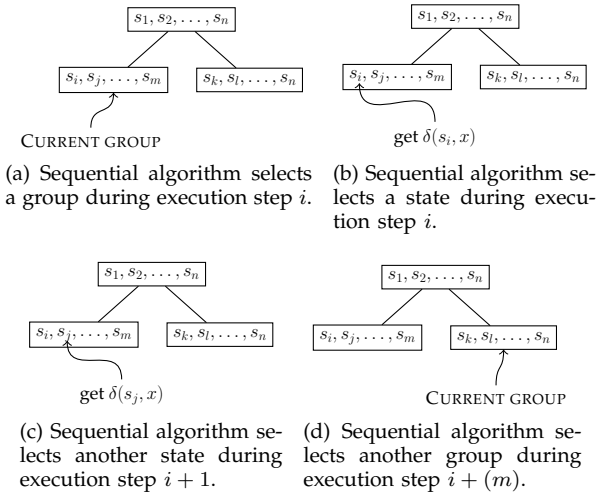
2.3 The CUDA Programming Model

Compute Unified Device Architecture (CUDA) is NVIDIA's parallel computing architecture that combines software and hardware architectures. We first present an overview of the CUDA hardware model.

At the hardware level, a CUDA capable GPU processor is a collection of *multiprocessors* (SMX), each having a number of *processors*. Each multiprocessor has its own shared memory which is common to all its processors. It also has a set of 32-bit (or 64-bit depending on the card) registers, texture memory (a read only memory for the GPU), and constant (a read only memory for the GPU that has the lowest access latency) memory caches. In any given cycle, each processor in the multiprocessor executes the same instruction on different data and so a multiprocessor is a *single instruction multiple data* (SIMD) processor. Communication between multiprocessors can be achieved through the *global device memory*, which is available to all the processors in all multiprocessors.

From the programmer's point of view the CUDA model is a collection of threads running in parallel, with a collection of threads, called a *warp*, running simultaneously on a multiprocessor. The warp size can vary according to the GPU. The programmer decides on the number of threads to be executed. If the number of threads is more than the warp size then these threads are time-shared internally on the multiprocessor. At a given time, a *block* of threads runs on a multiprocessor. The maximum number of threads in a block can vary according to the underlying GPU. However, multiple blocks can be assigned to a single multiprocessor and their execution is again time-shared. The collection of blocks for a single program is called a *grid* and the maximum number of grids can vary according to the GPU.

All threads of all blocks executing on a single multiprocessor divide its resources equally amongst themselves. Each thread executes a piece of code called a *kernel*. The kernel is the core code to be executed on a multiprocessor. Upon execution, thread t_i is given a unique ID and during execution thread t_i can access data residing in the GPU by using its ID. Since the GPU memory is available to all the threads, a thread can access any memory location. This allows programmers to interpret a device as a Parallel Random Access Machine (PRAM) architecture through the usage of global device memory. However, the performance improves with the use of shared memory (which can only be accessed by threads within a block), as such memory can be accessed faster than the global device memory. During GPU computation the CPU can continue to operate. Therefore the CUDA programming model is a hybrid computing model in which a GPU is referred as a co-processor (*device*) for the CPU (*host*).



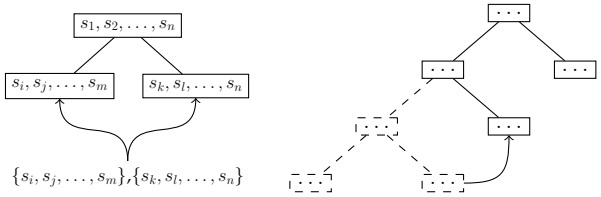
(a) Sequential algorithm selects a group during execution step i.

(b) Sequential algorithm selects a state during execution step i.

(c) Sequential algorithm selects another state during execution step $i+1$.

(d) Sequential algorithm selects another group during execution step $i+(m)$.

Figure 4: Steps of sequential algorithm overview. Note only current states are illustrated.



(a) Information about how states are split is kept by a UIO tree

(b) Algorithm back tracks on a UIO tree.

Figure 5: Use of tree structure while constructing a UIO-tree. Note only current states are illustrated.

3 HIGH-LEVEL DESIGN

In this section we present the high-level design of the proposed massively parallel (P-UIO) algorithm for deriving UIOs from FSMs. We start by discussing the design decisions made in developing the P-UIO algorithm and then provide an overview of the P-UIO algorithm. In Section 4 we provide additional information regarding the data structures used.

3.1 Parallel design: From UIO-Tree to Sorting

In the proposed parallel algorithm, we aimed to address several bottlenecks that we may encounter while using naïve UIO tree construction algorithms.

1) Sequential process: A naïve sequential UIO tree generation algorithm iterates over a UIO tree T and would process this tree node-by-node. Here each node is associated with a group and the same input is applied to each state in a group (since these states have yet to be distinguished). (Figure 4).
2) Memory Requirements: During UIO tree computation, all portions of the UIO tree would be kept in memory since:
   - It includes information about how the states are split (Figure 5a) and
   - It makes it possible to back-track when required (Figure 5b).

In developing a massively parallel approach, one might construct a UIO-tree and choose to have a thread t_i process a single node of a UIO-tree. The thread t_i would process all of the data associated with a node [12], [15] (Figures 7a). However, a node can have many current and initial states and for an FSM with n states, a node is associated with data whose size is of $O(n)$. Although the maximum number of states associated with a node reduces as the depth of the tree increases, the rate at which this happens will vary between FSMs (Figure 7b). As a result, an approach that directly represents the UIO tree may not scale well for very large FSMs.

These are crucial obstacles in designing a scalable UIO generation algorithm. In order to ease these issues, we need to devise a scalable alternative approach which demands less memory, can be parallelised, and can be used to derive UIO sequences. We now explain how this

can be achieved.

s_1	s_2	s_3	s_4	s_5	...	s_n
o_2	o_2	o_1	o_3	o_3	...	o_2
o_1	o_1	o_1	o_2	o_1	...	o_2
⋮	⋮	⋮	⋮	⋮	⋮	o_1
o_1	o_2	o_1	o_2	o_2	...	o_1

(a) Output sequences

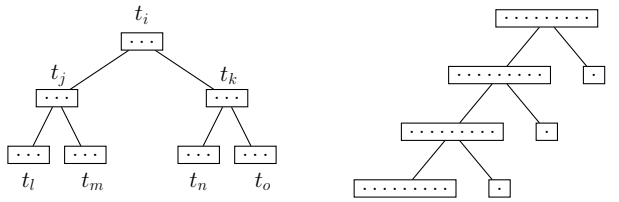
s_3	s_9	s_1	s_7	s_{11}	...	s_4
o_1	$o_1 \neq o_2$	$= o_2$	o_2	o_2	...	o_3
o_1	$o_1 = o_1$	$\neq o_2$	o_2	o_2	...	o_3
⋮	⋮	⋮	⋮	⋮	⋮	o_3
o_1	o_1	o_1	o_1	o_1	...	o_3

(b) Revealing distinguished states by considering sorted output sequences

Figure 6: The use of sorting while finding distinguished states.

Consider Figure 6a, which gives output sequences produced by the n states of an FSM in response to an input sequence $\bar{x}$. Input sequence $\bar{x}$ uniquely distinguishes a state if and only if the output sequence produced by this state is unique. Thus, we can use an approach that finds columns in which the output sequences are unique. We can re-formalise this problem as follows: *We are given a set of sequences of the same length and want to find the different sequences*. This problem can be solved by sorting the output sequences: after sorting, if the column being considered is different from its neighbouring columns then it is unique (Figure 6b). A benefit of this is that sorting can be parallelised and can be efficiently performed by GPUs [19]. Thus, in the P-UIO algorithm we used an approach based on sorting in order to determine which states have been distinguished.

In order to be able to use sorting we need to introduce an alternative formalisation for constructing UIO sequences. Rather than represent the problem in terms of UIO-trees, we use what we call *input output vectors*; later we will see how we can base a scalable parallel UIO generation algorithm on this formalisation.



(a) Each thread processes a node of a tree concurrently.

(b) An unbalanced UIO tree with very large groups.

Figure 7: Different views for parallelism (a),(b) on a UIO tree and an illustration for an unbalanced UIO-tree (c).

Definition 3.1: An input/output vector (IO-vector) V for an FSM $M = (S, X, Y, \delta, \lambda)$ with n states is a vector with n elements such that: For state $s_i \in S$ there exists an element v in V that is associated with *initial state* s_i, a *current state* $s_c \in S$, an input sequence $\bar{X}(v)$ such that $\delta(s_i, \bar{X}(v)) = s_c$, and an output sequence $\bar{O}(v)$ such that $\lambda(s_i, \bar{X}(v)) = \bar{O}(v)$. We will say that an IO-vector is *homogeneous* if every pair of elements that have the same output sequence also have the same current state.

We are interested in whether an IO-vector is homogenous since there is no value in extending such an IO-vector with further input: if two initial states s and s' have not been distinguished in this IO-vector (they have the same output sequences) then they are mapped to the same current state and so cannot be distinguished by further input. Thus, whenever the search for UIOs finds a homogenous IO-vector it will back-track.

In UIO generation, we will 'evolve' the elements of an IO-vector and will do so in a manner that is consistent with the notion of a UIO. We sort the output sequences to determine whether the states corresponding to two elements have been distinguished: two elements share the same output sequence if they have not been distinguished. In each iteration, for each output sequence that appears in the current IO-vector the algorithm chooses a next input to use. Let us suppose that element v is associated with initial state s_i and current state s_c. If the next input to apply in v is x then v is said to *evolve* to a new element v' with input x ($evolve(v) = (x, v')$) and we have that v' is associated with initial state s_i, current state s'_c, input and output sequences $\bar{X}(s_i)x/\bar{O}(s_i)o$ such that $\delta(s_c, x) = s'_c$ and $\lambda(s_c, x) = o$. The P-UIO algorithm will ensure that given an IO-vector V, if elements v and v' have the same associated input/output sequence then the process of evolving V will lead to the same next inputs in v and v'. As a result, for any pair of elements v and v' such that $\bar{O}(v) = \bar{O}(v')$ we have that: if $evolve(v) = (x, v'')$ and $evolve(v') = (x', v''')$ then $x = x'$. The following shows how IO-vectors are related to UIOs.

Lemma 3.1: Let us suppose that V is an IO-vector for M and that V has an element v whose initial state is s. If there does not exist $v' \in V \setminus \{v\}$ such that $\bar{O}(v) = \bar{O}(v')$ then $\bar{X}(v)$ is a UIO for s.

The proposed algorithm iterates over an IO-vector and ideally obtains what is called a UIO-vector.

Definition 3.2: An IO-vector V is said to be a *unique input output vector* (UIO-vector) if for any pair of elements $v, v' \in V$ with $v \neq v'$ we have that $\bar{O}(v) \neq \bar{O}(v')$.

Note that an IO-vector may not evolve into a UIO-vector. The reason for this is that in evolving an IO-vector the same input is applied to all elements that have the same output sequence. The problem here is that, for example, an FSM might have UIOs for all states but have a pair of states s, s' such that the UIOs for s and s' start with different inputs: such a scenario cannot be captured by an IO-vector. Consequently in order to construct UIOs one may need to construct a set of IO-vectors.

Definition 3.3: A set $\mathcal{V} = \{V_0, V_1, \ldots V_\kappa\}$ of IO-vectors

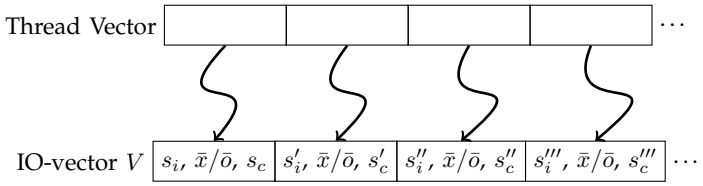


Figure 8: An illustration for a IO-vector. Each element is associated with an input sequence an initial state, and a current state.

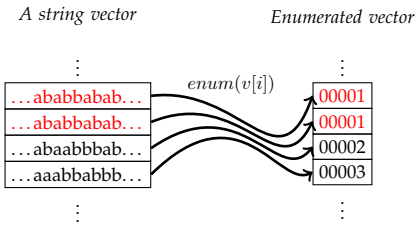


Figure 9: An illustration for enumeration. Each string is compacted to another string. Note that the enum function produces same values when given the same input string (red coloured texts).

is a *full set* for FSM M with state set S if for all $s \in S$, there exists an IO-vector $V' \in \mathcal{V}$ that has an element v whose initial state is s and whose input sequence $\bar{X}(v)$ is such that $\bar{X}(v)/\bar{O}(v)$ is a UIO for s.

The following is an immediate consequence of the definition of a full set.

Lemma 3.2: If FSM M has a full set $\mathcal{V}$ of IO-vectors then this defines UIOs for all states of M.

Importantly, an element of an IO-vector contains all information related to the evolution from a single state including the input/output sequence, initial and current states. This representation allows us to have a one-to-one correspondence between threads of the GPU and the elements of an IO-vector (Figure 8), overcoming the issue we had with UIO-trees where if a thread t_i processes a node then t_i considers $O(n)$ states.

However, note that if we insist on keeping input/output sequences within elements then each thread will process a whole input/output sequence during every iteration. As a result, the memory used by a single thread will increase and this may reduce the scalability of the algorithm. In order to avoid this, we devised an approach in which the input sequence associated with an element is kept elsewhere (not in the element). This is not problematic for inputs: when evolving an IO-vector we do not need to know about the previous inputs. However, we need to determine which elements of an IO-vector must be evolved using the same input. This suggests that threads should consider all the output sequences observed so far.

We addressed this problem as follows. Instead of keeping/sorting all output sequences observed, each element will keep a unique *representation* of an output sequence, the aim being to reduce the amount of data stored. In order to achieve this, an output sequence $\bar{o}$ is represented by an *enumeration* ($enum(\bar{o})$) that assigns a unique representation (number) to $\bar{o}$. For example, if we reach a point where only two output sequences have been observed then we could simply use the numbers 0 and 1. In the next section we describe how enumeration was done.

We will see that one important property of enumeration is that the equality relation over strings is preserved.

Lemma 3.3: Given $\bar{o}, \bar{o}', \bar{o}'' \in Y^\star$, $enum(\bar{o}) \neq enum(\bar{o}')$ if and only if $enum(\bar{o}\bar{o}'') \neq enum(\bar{o}'\bar{o}'')$.

We also have the following results.

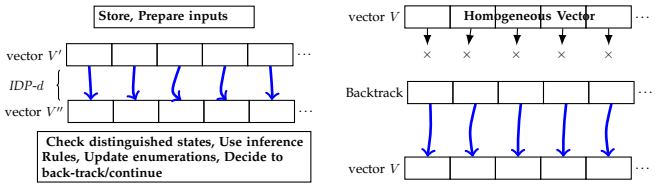


Figure 10: Overview of the steps taken in one iteration of P-UIO algorithm: the algorithm keeps evolving elements through the IDP. Before each iteration the algorithm stores the current data and chooses inputs to be applied. After each IDP, the algorithm checks for distinguished elements, applies inference rules, updates enumerations, and decides to continue or not. If the vector is homogeneous then it back-tracks.

Lemma 3.4: Let $v \in V$ be an element of an IO-vector associated with initial state s. If there does not exist $v' \in V \setminus \{v\}$ such that $enum(\bar{O}(v)) = enum(\bar{O}(v'))$ then s is distinguished from all other states of M.

Lemma 3.5: Let v and v' be two elements of V associated with initial states s, s' respectively. If $enum(\bar{O}(v)) = enum(\bar{O}(v'))$ then s and s' are not separated and if $enum(\bar{O}(v)) \neq enum(\bar{O}(v'))$ then s and s' are separated.

We can thus use the enumeration function to reduce the size of the information stored regarding the output sequences observed. It will be possible to define an enumeration function such that the size of the representation of the output sequence will be no larger than $\log(n)$ since it takes $\log(n)$ space to represent the integers from 0 to $n-1$ and there can be at most n different output sequences. Further, this information will allow the algorithm to determine which states have not been distinguished in constructing an IO-vector (so must be followed by the same input) using space of size no more than $\log(n)$ (Figure 9). As a result, the proposed approach will satisfy our requirements regarding GPU memory.

3.2 An overview of the P-UIO algorithm

In this section, we provide an overview of the P-UIO algorithm.

The P-UIO algorithm receives an FSM and positive integers d and ℓ and computes UIOs. The loop iterates until either (1) UIOs have been found for all states or (2)

the algorithm cannot back-track. The overall algorithm is like a depth-first search that, in an iteration of the main loop, increases the length of the input sequence being considered by d. We could simply increase the input sequence length by 1 in each iteration; there would then be no need to introduce the parameter d. However, as we will see later, this choice could lead to more frequent (relatively slow) memory transfers between the GPU and CPU to store the current data (for back-tracking).

If the overall depth of the process is greater than or equal to a bound ℓ at the end of an iteration of the main loop then the algorithm back-tracks and stores the elements that are associated with UIOs (those with a unique enumeration of output sequence). Therefore, the algorithm stores UIOs for M when they are found.

The algorithm can return UIOs of length greater than ℓ if d is not a factor of ℓ: if $kd < \ell \leq (k+1)d$ for integer k then the main loop back-tracks if the overall depth reaches $(k+1)d$. In addition, longer UIOs can be returned if the inference rules are used. The algorithm has the following phases in every iteration of the main loop.

- Phase 1: Apply the iterative deepening process.
  - Store the current IO-vector (to allow back-tracking).
  - Choose inputs for the next iteration of the deepening process. In this work we selected inputs randomly in such a way that no input sequence is applied to an IO-vector twice.
  - Apply an *iterative deepening process* (IDP) on the elements of the IO-vector, increasing the depth by d. The inputs used to evolve elements during the IDP phase are used *in-order*. By in-order we mean that elements that are associated with the same output sequences evolve with the same inputs. This ensures that if the same output sequences have been observed from two states s and s' then the same next input sequence is applied. As explained earlier, for reasons of efficiency the algorithm stores (and so compares) enumerations of the output sequences and not the output sequences themselves.
- Phase 2: Gather the outcome of IDP.
  - Find elements that define UIOs by sorting; such an element has a unique enumeration (of the output sequence).
  - By using inference rules, find additional UIOs.
  - Update the representations of output sequences.
- Phase 3: Decide to continue.
  - Continue from step 1 if some states do not have known UIOs, the upper bound (ℓ) on UIO lengths has not been reached, and the IO-vector is not homogeneous.
  - Backtrack if the upper bound on UIO lengths has been reached and not all UIOs are generated or the vector is homogeneous.
  - Terminate if the algorithm has generated a UIO for each state or it is not possible to continue or back-track.

Figure 10 describes the approach.

For a given state $s \in S$ we may need to consider every possible input sequence whose length is below the upper-bound ℓ and so the P-UIO algorithm is an exponential algorithm.

The following is immediate from the definition of the termination condition of the algorithm.

Theorem 3.1: Let us suppose that the P-UIO algorithm receives an FSM M and inputs d and ℓ. The P-UIO algorithm terminates with success if every state of M has a UIO sequence of length at most ℓ.

Note that this is not an 'if and only if' result since the use of d and inference rules allows the P-UIO algorithm to return UIOs of length greater than ℓ.

4 Low-level design

In the previous section we provided a high-level overview of the proposed algorithm. However, there is a need to map this to a structure that can be implemented using GPUs. In this section we give these low-level design details of the P-UIO algorithm. Although the P-UIO algorithm consists of only a few steps, in order to obtain high performance from the GPU, one has to consider the following design principles.

1) *Minimise global memory transactions*
2) *Maximise number of parallel threads per block*
3) *Prevent thread divergence*

Recall that in designing the P-UIO algorithm, our objective is to realise a one-to-one correspondence between the states of the FSM and the threads of the GPU. Therefore, we want to maximise the number of threads used in a block. However, this has implications regarding the use of shared memory on a GPU as shared memory usage limits the number of threads in a block.

Therefore, the P-UIO algorithm we did not use shared memory but instead we tried to minimise the global memory transactions latency. As previously reported [45] global memory transaction latency can be hidden by using (1) many threads and (2) supporting *coalesced memory access* in which the threads in a block access global memory in a manner that allows the GPU to bundle a number of memory accesses into one memory transaction. The principle of memory coalescing is similar to the *cache line* principle of a CPU, in which a cache line is either completely replaced or not at all. Even if only a single data item is requested, the entire line is read so whenever a neighbouring item is subsequently requested, it is already in the cache.

Recall that threads from a block are grouped into warps for execution on a CUDA core and threads within a warp must follow the same execution trajectory (otherwise we have thread divergence). That is, all threads must execute the same instruction at the same time. In order to satisfy this constraint, in the P-UIO algorithm we avoided if-else structures where possible.

Type of Memory	Data Structure
Texture Memory	D_FSM, $D\_inferenceRules$
Global Memory	D_inputs, $D\_outputs$, D_states and temporal vectors.
Registers	temporary variables
Local Memory	NA
Shared Memory	NA

Table 1: Summary of the different types of memory used in the P-UIO algorithm.

We now discuss how the high-level P-UIO algorithm was refined for use with GPUs. In order to perform coalesced global memory access, in the P-UIO algorithm we use several structures to represent an IO-vector V; these structures are to be kept in the global memory of the GPU (as opposed to the local memory of a thread) and hold the following information.

1) The D_states vector holds the relationship between initial and current states: given initial state s_i, $D\_states[i]$ is the corresponding current state.
2) The D_inputs vector holds the inputs that will be applied during the next step of IDP.
3) The $D\_outputs$ vector holds *output data*: enumerations of output sequences observed from initial states. It therefore provides information regarding which states have been distinguished (split).
4) The D_FSM vector holds the transitions of the underlying FSM.
5) $D\_inferenceRules$ holds the unique precedence information for each state.

Table 1 summarises the memory management. The FSM and inference rules were associated with the texture memory. The vectors used to construct UIOs (such as D_inputs, $D\_outputs$, D_states) were declared as global memory. Variables used for the computation were automatically associated with registers by the compiler. In developing the P-UIO algorithm we did not use local and shared memory.

As explained in Section 3.2, the P-UIO algorithm has a main loop that has three phases:

1) In the first phase IDP is applied to extend the depth by d (Phase 1 in the high-level description of the loop, described in detail in Section 4.1);
2) In the second phase, the outcome of IDP is analysed (Phase 2 in the high-level description, described in detail in Section 4.2); and
3) In the final phase the algorithm decides whether to continue or back-track (Phase 3 in the high-level description, described in detail in Section 4.3).

The algorithm is summarised in Algorithm 1. Here lines 3-5 correspond to Phase 1 in the high-level description (Section 3.2); lines 6-9 correspond to Phase 2; and lines 10-11 correspond to Phase 3. The shading shows the steps that are carried out in parallel.

Algorithm 1: Parallel UIO construction Algorithm. Highlighted lines are executed in parallel.

Input: A deterministic completely specified FSM with n states, p inputs and r outputs and positive integers d and ℓ.
Output: A set of input sequences
begin
1 $id \leftarrow 0$
2 Initialise D_states, D_inputs, $D\_outputs$,D_FSM, and $D\_inferenceRules$ vectors.
3 **while** *not all UIOs are computed and the algorithm can continue* **do**
4 Store current vectors and id value.
5 Construct inputs to be applied and update the id value.
`// Apply iterative deepening`
6 Apply iterative deepening process until depth d.
`// Gather outcomes of IDP.`
7 Sort $D\_outputs$ vector and retrieve new distinguished states.
8 **while** *New UIOs are found* **do**
9 Apply inference rules
10 For each state, update enum() values.
`// Decide to continue`
11 **if** *Algorithm cannot continue* **then**
12 Back-track repeatedly until the algorithm can continue or the root is reached.

4.1 Applying the Iterative Deepening Process

Before IDP begins, id and vectors are stored in CPU memory for back-track (Line 4) and then the input vector is generated by a *parallel random combinator generator* (PRCG) (Line 5). This procedure receives an integer value id, number of states n, alphabet X, the D_inputs vector, and iterative deepening parameter d. PRCG first calls a kernel called random number generator (RNG). RNG receives n, d, and p and it returns an integer value v in the range $[0, p^{nd}]$. Then the algorithm checks if it can increment[2] id, if so the PRCG calls another kernel (the Fill kernel). The Fill kernel receives v, X, and the D_{inputs} vector as its parameters and fills the D_inputs vector with the representation of v in base p. Otherwise, if the algorithm cannot increment id then it calls the RNG kernel and repeats the process. Later the algorithm enters IDP (Line 6).

During an iteration (say the jth iteration) of IDP, the Host calls several kernels one after another. The first kernel to be called is the Apply kernel. During execution of the Apply kernel a thread t_i reads the ith value on D_states vector and gathers the current state. If the value is -1 the thread halts. Otherwise thread t_i reads the ith value from the $D_{outputs}$ vector ($D_{outputs}[i]$) and retrieves input $x = D\_inputs[(j * n) + D_{outputs}[i]]$ from the D_{inputs} vector. Note that as an input is computed based on the output, at each iteration the same input is applied to all states for which the values read from the $D_{outputs}$ vector are identical. Then, the thread uses the D_FSM vector to determine the next state $s \in S$ and output $o \in Y$. Afterwards, it writes s to the D_{states} vector and

2. Note that the algorithm increments id if v has not been selected for the IO under consideration.

it concatenates o with $D_{outputs}[i]$.

The next step is to update the $D_{outputs}$ vector. To achieve this, we follow a similar procedure which is applied to check if new pairs of states are split. This procedure is explained in the following section. But in summary we apply two steps: sort the $D_{outputs}$ vector, write integer values (starting with 0) to elements of the $D_{outputs}$ vector so that two elements of $D_{outputs}$ vector are identical if and only if they receive same integer value. As there are at most n different possible output sequences, the number assigned to an element of the $D_{outputs}$ vector is between 0 and n.

During IDP, for a state s_i a thread t_i will normally read the corresponding indexes on D_states, D_inputs, $D\_outputs$ and D_FSM, many times. Although the reads and writes on the D_states and $D\_outputs$ vectors are coalesced, transactions on D_inputs are not. The index values in the D_inputs vector depends on the data retrieved from the $D\_outputs$ vector. Moreover, since the host can write the FSM transition structure to D_FSM once and the kernels can read this FSM structure many times, the D_FSM vector is stored in the texture memory and so coalescing is not an issue. On the other hand note that IDP does not allow thread divergence. That is, all threads in a warp will process the same instruction of a kernel.

4.2 Gathering outcomes of IDP

Once IDP ends, we need to check if new pairs of states are split. This is done through a *parallel stable sorting* (Line 7). The sorting algorithm gathers vector $D\_outputs$ and a vector (D_keys) which holds the relative orders of items of D_states (the initial state information). It then sorts the states according to $D\_outputs$[3] (Figure 11). The results of the sort reveals states that are distinguished from all other states (singleton states) and pairs of states that produce the same output sequences.

In order to achieve this a temporary vector ($D\_singletons$) is used. After receiving $D\_keys, D\_outputs$ and $D\_singletons$ a thread (thread t_i) selects a single (ith) item of the $D\_outputs$ vector and compares the enumeration of the output sequence to those of the neighbouring values (the enumeration of the output sequence read from the $i+1$th and $i-1$th locations of the $D\_outputs$ vector). If the ith value is different from both of these values then the thread reads the initial state information from the D_keys vector and stores it in the $D\_singleton$ vector. In order to determine which states are split, another temporary vector called the D_groups vector is used as follows: a thread again selects a single (ith) item of the $D\_outputs$ vector and reads the *index* of the initial state information from the D_keys vector only if one neighbouring output data is the same as that of the ith item of the $D\_outputs$ vector (not both). As a result of this process, for each group of states with the same output data we have two values in the D_groups vector indicating the starting and ending indexes of the initial states in a group from the D_keys vector (Figure 12). Note that the process of finding singletons and groups of states can cause thread divergence, which may prevent threads in a warp from executing concurrently.

3. Note that after the sort, the information in $D\_outputs[i]$ may not belong to initial state s_i.

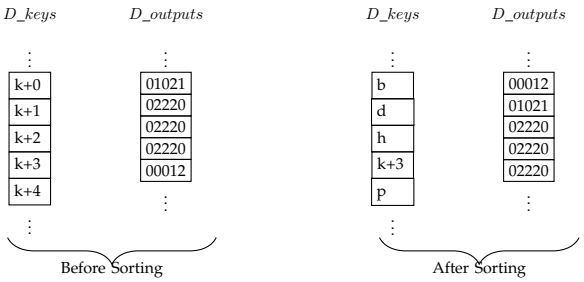


Figure 11: An illustration for sorting. Stable sorting algorithm recives $D\_outputs$ vector and D_keys vector as satellite information.

After singletons have been found, a kernel uses the inference rules to try to find UIOs for other states (Lines 8–9). In order to achieve this, the kernel receives the list of singletons found and the $D\_InferenceRules$ vector. Each thread selects one state from the $D\_singletons$ vector and finds its unique predecessors from the $D\_InferenceRules$ vector. Note that similar to the D_FSM vector, the host can write the $D\_InferenceRules$ vector once and the kernels can read this data many times, therefore the $D\_inferenceRules$ vector is stored in texture memory and so coalesced memory access is not an issue. Moreover, since the algorithm should also consider unique predecessors of fresh states, the kernel may be called by the Host until it reaches a point where no fresh states are found.

Once singletons and groups have been revealed, the algorithm assigns unique integers (beginning from 0) to singleton states and groups of states and this defines the *enumeration* of each corresponding output sequence (Line 10). The algorithm then updates representations (i.e. $enum(\bar{O}(s))$) of states. A thread t_i is assigned to a group S' and generates a unique integer value ($\kappa_{S'} = i$). Then for all $s \in S'$, t_i retrieves the initial state information from the D_keys vector and it writes $\kappa_{S'}$ to $D\_outputs$ (Figure 12).

4.3 Checking Termination Conditions

After enumerations are computed, the algorithm checks if UIOs of some states have been found in the current level, if so the algorithm stores the corresponding input sequences in CPU memory (Lines 11–12). Afterwards the algorithm decides what to do next. If not all input sequences of length less than ℓ have been applied and either the underlying IO-vector is homogeneous or it has reached the upper bound ℓ then the P-UIO algorithm back-tracks. If it back-tracks, all the data (except data

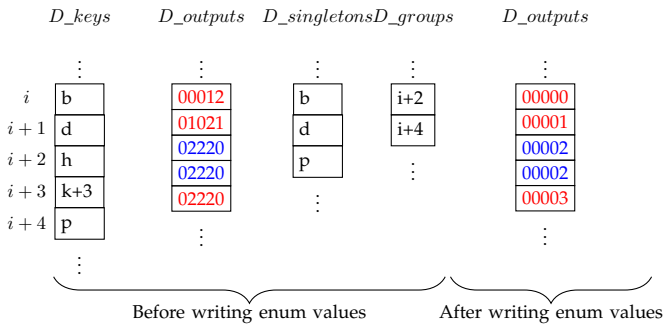


Figure 12: An illustration for enumeration. Each string is compacted to another string. Note that for same strings enum function produces same values (red coloured texts).

related to singletons) computed in the current iteration of the main loop is discarded and the previous data, which resides in CPU memory, is brought to GPU memory. The P-UIO algorithm then continues to execute. Otherwise, if a UIO has been found for every state then the algorithm ends execution. If neither of these conditions holds then the algorithm continues to execute with current D_states and $D\_outputs$ vectors.

Note that after each IDP the algorithm stores the current data in CPU memory. As memory transactions between CPU and GPU are expensive, it is good practice to reduce the number of such transactions. As a result, it makes sense to select relatively large values of d. If we pick $d = 1$ then each time we increase the input sequence length by 1 we need to send data back to the CPU and this will reduce the performance of the algorithm. In the next section we report on the results of experiments that show how the value of the parameter d affects the performance of the algorithm.

4.4 Example

We now show the execution of the P-UIO algorithm using an example. Consider the FSM given in Figure 3a. Let us suppose that M_2, $d = 2$ and $\ell = 5$ are provided to the P-UIO algorithm as parameters. Then the algorithm first sets $id = 0$, then initiates vectors (an IO vector).

$$D_{states} = \langle s_1, s_2, s_3, s_4 \rangle$$
$$D_{outputs} = \langle 0, 0, 0, 0 \rangle$$

It then stores the values of id and the vectors to CPU memory. Afterwards it randomly generates an input sequence, increments id, and sets $id = 1$. Let us suppose that

$$D_{inputs} = \overbrace{x_2x_1x_2x_1}^{\text{possible inputs for 0th iteration}}\overbrace{x_1x_1x_1x_2}^{\text{possible inputs for 1th iteration}}$$

Note that the length of D_{inputs} is dn since $n = 4$. The P-UIO algorithm then evolves the elements of the vectors as follows.

The 0th iteration: as all output values are 0, the Apply kernel picks the element at index $0 * 4 + 0$ of the D_{inputs} vector (x_2) and the vectors become

$$D_{states} = \langle s_2, s_3, s_2, s_1 \rangle$$
$$D_{outputs} = \langle o_2, o_2, o_2, o_2 \rangle$$

The 0th iteration: once the $D_{outputs}$ vector has been sorted and updated, the vectors become

$$D_{states} = \langle s_2, s_3, s_2, s_1 \rangle$$
$$D_{outputs} = \langle 0, 0, 0, 0 \rangle$$

The 1st iteration: as all the outputs in $D_{outputs}$ are 0, during the Apply kernel the element at index $1 * 4 + 0$ of the D_{inputs} vector is picked (x_1) as input and the vectors become

$$D_{states} = \langle s_3, s_4, s_3, s_2 \rangle$$
$$D_{outputs} = \langle 0o_1, 0o_1, 0o_1, 0o_2 \rangle$$

The 1st iteration: after the $D_{outputs}$ vector is sorted and updated the vectors become

$$D_{states} = \langle s_3, s_4, s_3, s_2 \rangle$$
$$D_{outputs} = \langle 0, 0, 0, 1 \rangle$$

After the second iteration (since $d = 2$) the algorithm moves to the next step. Now as the initial state s_4 has a different output, the algorithm concludes that x_2x_1 in an input sequence that distinguishes state s_4 from any other states. Later it proceeds with the inference rules given in Figure 3b and finds input sequences for other states as $s_1 = x_1x_2x_1x_2x_1$, $s_2 = x_2x_1x_2x_1$ and $s_3 = x_1x_2x_1$. Since a UIO has been found for each, the algorithm terminates.

5 Empirical Study

In this section we present the results of our experiments. We used an Intel Core 2 Extreme CPU (Q6850) with 8GB RAM and 64 bit Windows Server 2008 R2 operating system. The GPU computing approach (separately) used three NVIDIA GPUs: a TESLA K40, a TESLA c2070, and a TESLA c1060. In the experiments, we evaluated the methods by investigating the average time to construct UIOs for FSMs and the average length of the UIOs constructed. For the P-UIO method, we used $d = 40$ as the default value. However, this value affects the performance of the algorithm and so we also performed experiments with different d values.

We used several sets of FSMs, described below. Moreover, we also set the upper-bound on the length of UIOs as $\ell = n^2$ where n is the number of states. This value was chosen since, as noted earlier, it is an upper bound on the sum of the lengths of the sequences in a characterisation set (assuming a sensible algorithm has been applied to generate the characterisation set). In Naik's algorithm, there is no upper-bound on the length of UIO's. However we believe that this has at least two drawbacks. First, very long UIOs will typically be of little value when computing test sequences; instead we can use alternative approaches (with polynomial upper bounds on size) such as characterising sets. In addition, if the FSM does not possess UIOs for all of its states, then Naik's algorithm constructs the complete UIO-tree

in the worst case. Note that some FSMs did not have UIOs of length ℓ or less for all states and these were also discarded; later we report on this.

5.1 FSMs used in the experiments

5.1.1 The FSMs in SUITE I

The FSMs in this suite were designed to investigate the performance of the methods under varying number of states. We fixed the number of inputs and outputs to be $p = 2$ and $r = 2$.

The FSMs in this class were generated as follows. First, for each input x and state s we randomly assigned the values of $\delta(s, x)$ and $\lambda(s, x)$. After an FSM M was generated we checked its suitability as follows. We checked whether M was strongly connected and minimal. If the FSM failed one or more of these tests then we omitted this FSM and produced another. Consequently, all FSMs were strongly connected and minimal.

By following this procedure we constructed 100 FSMs with n states, where n is a power of 2 and $n \in \{64, 128, \ldots, 524288, 1048576\}$. In total we constructed 1500 FSMs for the first test suite.

5.1.2 The FSMs in test SUITE II

These FSMs were used to explore the effect of the size of the output alphabet. We fixed the number of states to be 1024 and constructed 100 FSMs with each of the following sizes i/o of input/output alphabets: $i/o \in \{128/2, 128/128, 128/256\}$. As a result there were 300 FSMs in SUITE II.

5.1.3 The FSMs in test SUITE III

While using randomly generated FSMs allowed us to perform experiments with many subjects and see how performance changes as the problem size increases, it is possible that FSMs used in practice differ from these randomly generated FSMs. We therefore complemented the experiments with case studies from the ACM/SIGDA benchmarks, which is a set of test suites (FSMs) used in workshops between 1989 and 1993 [46]. The benchmark suite has 59 FSMs, for circuits, obtained from industry.

The circuits were represented using the *kiss2* file format; a standard format devised by manufacturers [46]. In this format, inputs and outputs are represented as binary numbers, and states are represented as alphanumeric characters. For example a transition provided in *kiss2* file format $(s1, 11/10111000, s1)$ tells us that if input 3 is received when the FSM is in the state called $s1$ then there is no state change and output 184 is produced. Therefore, it is straightforward to obtain FSM specification from a circuit design written in the *kiss2* file format.

We used FSMs from the benchmark that were minimal and deterministic. We completed partial FSMs by introducing self loop transitions for missing transitions. Thus, for example, if there was no transition from state s with input x then a transition from s to s with input x and null output was added.

5.2 Results

In order to carry out these experiments for each FSM we computed UIO sequences using (1) Naik's UIO construction algorithm (implemented as given in [15]), and (2) the P-UIO algorithm. For a given method we constructed UIO sequences for each FSM in our pool.

5.2.1 Results of Experiments for FSMs in SUITE I

We present the mean timing results in Figure 13a. As expected, when the size of the FSM grows, the time required to construct UIOs increases. We observe that Naik's approach took less than three seconds on average to generate UIOs for FSMs with 512 states. For FSMs with 1024 states the time rises to 68.45 seconds, for FSMs with 2048 states the average time to construct UIOs is 1231 seconds. Therefore we did not process FSMs with more than 2048 states. These results suggest that the P-UIO algorithm can increase the scalability of Naik's algorithm by a factor[4] of 512.

The results show that when the NVIDIA TESLA K40 card was used, UIOs for very large FSMs (FSMs with 1 million states) could be constructed in less than two seconds (1626 msec on average). With the TESLA c2070 card the average time required increased to 3170 msec. and with the TESLA c1060 card the average time increased to 3658 msec. In Table 2 we provide the reduction in timings. The results for SUITE I indicate that P-UIO can be 11000 times faster then the existing UIO construction algorithm on average.

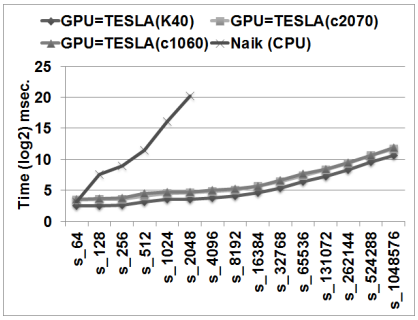
In Figure 13b we present the distribution of time spent by the P-UIO algorithm where Sort, Inference Rules, MemCpy, and Iterative Deepening stand for the average time spent for sorting, average time spent for finding new UIOs using inference rules, average time spent for memory transactions between the CPU and the GPU and the average time spent for IDP respectively.

The results suggest that most of the time required to construct UIO sequences was spent on sorting (averages vary between 37.6% and 45.01%) and we also observe that the use of inference rules took $25\% - 30\%$ of the time on average, which (compared to time spent on sorting) was not expected. Therefore we investigated the effect of using inference rules by counting the number of UIOs found using inference rules and the number of UIOs found during exploration. Figure 13d summarises the results.

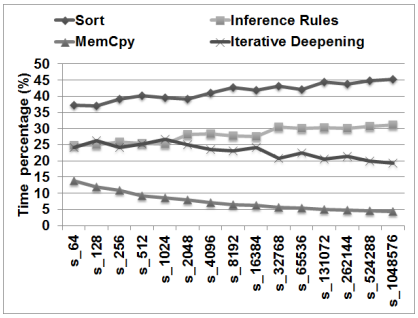
The results suggest that on average at least 80% of the UIOs were found using inference rules. These results justify the time spent on inference rules. In addition, the average percentage of time used for back-tracking (Memcpy) and iterative deepening reduces as the size of the FSM increase. This implies that as we increase the number of states, the utilisation of the GPU increases.

Figure 13c gives the results regarding UIO length. Note that the length of the UIOs returned by the P-UIO

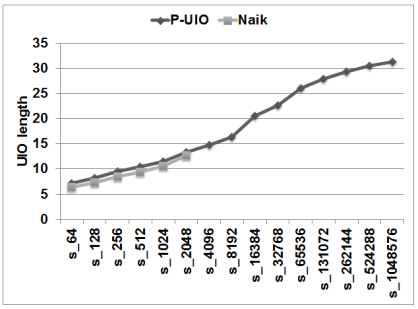
4. The P-UIO algorithm could process FSMs with 2^{20} states and Naik's algorithm could process FSMs with 2^{11} states hence we have $2^{20}/2^{11} = 2^9 = 512$



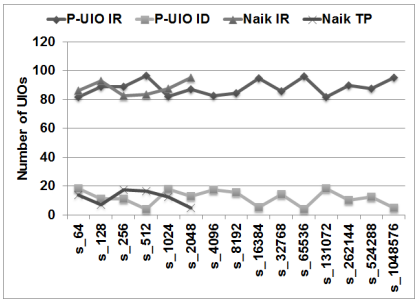
(a) Average time required to construct UIOs for FSMs in SUITE I



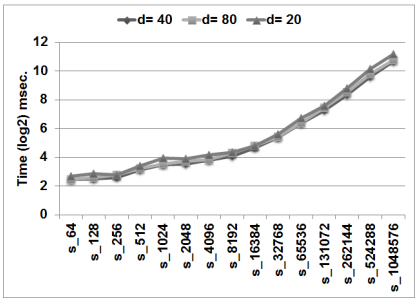
(b) Average time percentages, spent on different parts of the P-UIO algorithm for FSMs in SUITE I



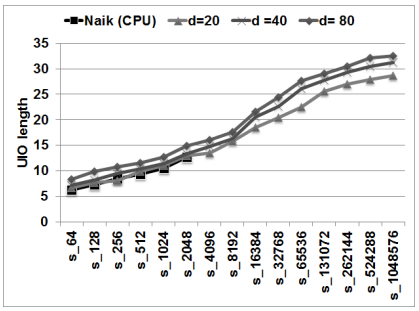
(c) Average lengths of UIOs derived by Naik and P-UIO algorithms for FSMs in SUITE I



(d) Effect of using inference rules. UIO sequences of more than 80% of the states are computed through inference rules.



(e) Time required to construct UIOs for FSMs in SUITE I with different iterative deepening parameter d.



(f) Length of UIOs derived from FSMs in SUITE I with different iterative deepening parameter d.

Figure 13: Results of experiments on test SUITE I.

	s_{64}	s_{128}	s_{256}	s_{512}	s_{1024}	s_{2048}	$Avg.$
T(Naik)/T(P-UIO$_{TESLA(K40)}$)	1.56	33.70	79.56	316.99	6001.64	104091.79	18420.88
T(Naik)/T(P-UIO$_{TESLA(K20)}$))	0.78	16.20	41.44	166.84	2956.47	49567.52	8791.54
T(Naik)/T(P-UIO$_{C1060}$))	0.76	15.60	34.00	126.79	2655.59	45454.93	8047.95
$Avg.$	1.04	21.83	51.67	203.54	3871.24	66371.41	11753.45

Table 2: The ratio of computation times.

algorithm does not depend on the underlying card and so we present the results obtained from the TESLA K40 GPU card. The results suggest that compared to the P-UIO algorithm, Naik's approach can find shorter UIOs (13% shorter on average). This result may be caused by the iterative deepening process. As the P-UIO algorithm iteratively deepens an IO-vector until it reaches depth d, it need not find the shortest UIOs. To investigate this we performed a set of experiments and repeated the tests on the P-UIO algorithm with different d values.

The results are presented in Figure 13e and Figure 13f. These results suggest that as we decrease the depth parameter (to $d = 20$), the time required to construct UIOs increases. This is because as we decrease d, the performance of the GPU reduces due to the frequent memory copy operations. However, the length of the UIO sequences reduces: when $d = 20$, the average difference between the length of UIO sequences constructed by the Naik and the P-UIO algorithms reduces to 6%. On the other hand, as we increase the iterative deepening parameter d to $d = 80$ again the time required to construct UIOs increases. This may be due to the fact that as we increase d, we also increase the amount of data that is sorted after IDP. Moreover, when we use $d = 80$ the length of the UIOs increase: Naik's algorithm generates UIOs that are 38% shorter compared to the P-UIO algorithm on the average.

During the experiments we observed that some FSMs did not have UIOs for every state. We observe that as the number of states increase the chance of generating FSM reduces. In order to generate 100 FSMs that have a UIO for each state we generated 232 FSMs when $n = 64$, 422 FSMs when $n = 128$, 628 FSMs when $n = 256$, 839 FSMs when $n = 512$, 1097 FSMs when $n = 1024$, 1265 FSMs when $n = 2048$, 1322 FSMs when $n = 4096$, 1493 FSMs when $n = 8192$, 1755 FSMs when $n = 16384$, 1959 FSMs when $n = 32768$, 2103 FSMs when $n = 65536$, 2395 FSMs when $n = 131072$, 2595 FSMs when $n = 262144$, 2797 FSMs when $n = 524288$ and 2903 FSMs when $n = 1048576$. Recall that if UIOs are not found for all states then one can instead use a characterising set that contains at most $n-1$ sequences of length at most $n-1$ and a characterising set can be found in polynomial time.

r/p	Naik (msecs)	P-UIO (msecs)	T(Naik)/ T(P-UIO)
128/2	222345	23.442	9484.89
128/128	136236	15.254	8931.16
128/256	126324	11.002	11481.91

Table 3: Average time to construct UIOs for FSMs in SUITE II and average increment in timings.

r/p	Naik	P-UIO	L(Naik)/ L(P-UIO)
128/2	10.82	17.27	0.62
128/128	10.27	11.35	0.90
128/256	11.45	11.27	1.01

Table 4: Average length of UIOs for FSMs in SUITE II.

5.2.2 Results of experiments for FSMs in SUITE II

The time required to construct UIOs for FSMs in SUITE II is given in Table 3. Throughout these experiments we set $d = 40$. As expected, as the number of outputs increases, the time required to construct UIOs decreases. We observe that one particular reason for this is that as the number of outputs increases the length of the UIOs derived from FSMs tends to reduce (Table 4). This is to be expected: as the number of outputs increases, the algorithms (Naik, P-UIO) have more opportunities to split states, hence the length of the UIOs reduce. However, we see that the performance of the P-UIO algorithm is far better than that of Naik's algorithm (9900 times faster on average).

5.2.3 Results of experiments for FSMs in SUITE III

The results are presented in Table 5 where we set $d = 40$. The time required to construct UIOs with Naik's algorithm and the P-UIO algorithm (with the Tesla K40 card) are similar for FSMs $dk27$, $bbtas$, $dk17$, and $dk15$. Moreover, for these FSMs the P-UIO algorithm is slower when Tesla C2070 and C1060 were used. However, as the FSMs get larger the time required to construct UIO with Naik's approach increases faster than that required by the P-UIO algorithm. As before, we also observe that the UIOs found are shorter when Naik's approach is used.

5.3 Threats to validity

This section briefly reviews threats to validity and how these were reduced. We consider threats to internal validity, construct validity, and external validity.

Threats to internal validity concern factors that might introduce bias. The main source of such threats is the tools used to run the experiments. The FSM generation tool has been used in a number of projects and was tested. The implementations of the two algorithms were carefully checked and also tested with a range of FSMs. To further reduce this threat, we also used an existing tool that checks if an input sequence is a UIO for the FSM. This tool was used to check all of the UIOs generated by the P-UIO algorithm and Naik's approach.

Another threat to internal validity concerns the random process employed while selecting input sequences: the order of selection may effect the performance of the algorithm. To investigate this factor, we repeated each experiment on Test SUITE III 100 times. The results are provided in Table 6. We observe that, except for the specification named $planet$, the variance of timing and length of UIOs are low, that is to say for this set of FSMs the random input selection process has limited effect.

Threats to construct validity reflect the potential for the measurements made to not reflect properties that are of interest in practice. The main focus of our study was the time taken to generate UIOs and, as a result, the scalability of the algorithm. We want FSM-based test generation techniques that scale to large FSMs and so scalability is important. Note that FSMs are likely to be particularly large when one cannot abstract out all of the data of a model, since we then obtain a separate state for each logical state of the model combined with each possible combination of values for the model's variables. However, to reduce the scope for threats to construct validity we also recorded the mean UIO length.

Threats to external validity concern our ability to generalise from the experiments. There is always such a threat to validity since we do not know the space of relevant FSMs and certainly have no good way of sampling from this. We reduced this threat by using a combination of randomly generated FSMs and FSMs from industry that are in a benchmark. We also varied the number of outputs and states.

5.4 Discussion

Recall that in Section 3 we observed that the P-UIO algorithm is an exponential algorithm; this cannot be avoided since determining the existence of UIOs is PSPACE-hard. As the length of the UIO sequences generated from the FSMs in SUITE I and SUITE II are not longer than the logarithm of the number of states, it appears that we have not found such long executions. However, it has been reported [12] that this is usual: the length UIO sequences are often no longer than the logarithm of the number of states of the FSM. Another important point is the need to select parameter d. The experiments revealed that when we select a value for d that is too large, the algorithm gets slower as the size of data to be sorted increases. However, if d is too small then this may decrease the GPU occupancy and increasing the traffic between the CPU memory and the GPU memory. Therefore, the parameter d should be selected carefully.

6 CONCLUSIONS

This paper explored the problem of constructing UIOs for very large FSMs. We proposed a new massively parallel algorithm that can construct UIOs for FSMs with millions of states. We presented the parallel design, issues encountered, and proposed solutions for the issues.

FSM Properties				Naik (CPU)		P-UIO (K40)		P-UIO (C2070)	P-UIO (C1060)
Name	$\|X\|$	$\|Q\|$	$\|Q\| * \|X\|$	L_{max}	T_{msecs}	L_{max}	T_{msecs}	T_{msecs}	T_{msecs}
$dk27$	2	7	14	6	12.65	8	12.26	25.92	27.91
$bbtas$	4	6	24	8	11.74	10	12.41	25.85	27.26
$dk17$	4	8	32	2	12.77	3	13.02	27.21	29.16
$dk15$	8	4	32	2	15.63	3	12.51	25.27	27.17
$ex7$	11	5	55	1	19.73	1	13.21	27.36	29.22
mc	5	15	75	1	22.27	1	15.11	21.82	23.71
$dk512$	2	15	105	4	29.63	5	17.25	24.44	26.41
$dk16$	4	27	108	3	28.35	3	19.22	28.13	30.95
$donfile$	6	18	180	4	12.34	6	18.71	26.67	28.34
$s386$	128	13	1664	5	53.73	6	14.14	27.62	29.26
$bbsse$	128	13	1664	4	49.84	4	15.33	29.32	31.52
$s1$	256	18	5210	1	31.94	1	17.62	24.72	28.26
$planet$	128	48	6144	4	37.01	14	17.55	22.53	26.88

Table 5: Results of Case Studies.

Name	L_{max}	T_{msecs}	$Avg(L_{max})$	$Avg(T_{msec})$
$dk27$	$7 \leq . \leq 9$	$11.33 \leq . \leq 12.78$	7.31	11.78
$bbtas$	$9 \leq . \leq 10$	$11.04 \leq . \leq 12.69$	9.22	11.74
$dk17$	$2 \leq . \leq 3$	$12.44 \leq . \leq 13.77$	2.25	13.15
$dk15$	$2 \leq . \leq 3$	$12.39 \leq . \leq 13.29$	2.36	12.99
$ex7$	1	$13.18 \leq . \leq 13.22$	1	13.19
mc	1	$15.17 \leq . \leq 15.28$	1	15.21
$dk512$	$4 \leq . \leq 6$	$16.01 \leq . \leq 17.51$	4.89	16.72
$dk16$	3	$18.88 \leq . \leq 19.75$	3	19.26
$donfile$	$4 \leq . \leq 6$	$17.62 \leq . \leq 19.01$	5.22	18.67
$s386$	$5 \leq . \leq 7$	$13.21 \leq . \leq 14.07$	6.64	13.88
$bbsse$	4	$14.93 \leq . \leq 15.58$	4	15.33
$s1$	1	$17.55 \leq . \leq 17.81$	1	17.68
$planet$	$6 \leq . \leq 14$	$17.01 \leq . \leq 18.77$	7.25	17.59

Table 6: Results of test SUITE III obtained after 100 consecutive runs.

The proposed algorithm has exponential worst time complexity. In order to evaluate the proposed algorithm, we performed an experimental study by comparing the proposed algorithm with a well-known UIO generation algorithm and investigated both the time required to construct UIOs and the lengths of the UIOs produced. In the experiments the P-UIO algorithm was able to handle FSMs with 1,048,576 states in under 2 seconds on average while the implementation of Naik's algorithm took 1231 seconds on average for FSMs with 2048 states. The two algorithms had similar performance for the benchmark FSMs but these FSMs were much smaller (at most 48 states) and there was a difference in performance for the larger benchmark FSMs.

There are several possible lines of future research. We plan to investigate massively parallel UIO generation algorithms for partial deterministic and nondeterministic FSMs. We also plan to investigate massively parallel algorithms for generating other types of sequences such as distinguishing sequences, characterising sets and checking sequences for complete / partial and deterministic / nondeterministic FSMs. There is also the question of whether guidance can be provided regarding the choice of d. Finally, there may be potential to adapt the proposed approach to problems regarding FSM inference.

7 ACKNOWLEDGEMENT

This research was supported by The Scientific and Technological Research Council of Turkey (TUBITAK) under grant no 1059B191400424 and by the NVIDIA corporation.

REFERENCES

[1] E. P. Moore, "Gedanken-experiments," in *Automata Studies*, C. Shannon and J. McCarthy, Eds. Princeton University Press, 1956.

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

[3] 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," *Communications, IEEE Transactions on*, vol. 39, no. 11, pp. 1604 –1615, nov 1991.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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*. Japan: Chapman and Hall, 1994, pp. 209–224.

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

[11] S. T. Vuong, W. W. 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.

[12] 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.

[13] 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.

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

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

[16] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *High performance computing–HiPC 2007*. Springer, 2007, pp. 197–208.

[17] L. Luo, M. Wong, and W.-m. Hwu, "An effective GPU implementation of breadth-first search," in *Proceedings of the 47th design automation conference*. ACM, 2010, pp. 52–55.

[18] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 117–128.

[19] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–10.

[20] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, ser. GH '07. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, pp. 97–106.

[21] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[22] J. E. Hopcroft, "An n log n algorithm for minimizing the states in a finite automaton," in *The theory of Machines and Computation*, Z. Kohavi, Ed. Academic Press, 1971, pp. 189–196.

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

[24] V. Jusas and T. Neverdauskas, "FSM based functional test generation framework for VHDL," in *18th International Conference on Information and Software Technologies (ICIST 2012)*, ser. Communications in Computer and Information Science, vol. 319. Springer, 2012, pp. 138–148.

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

[26] E. Brinksma, "A theory for the derivation of tests," in *Proceedings of Protocol Specification, Testing, and Verification VIII*. Atlantic City: North-Holland, 1988, pp. 63–74.

[27] 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.

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

[29] D. P. Sidhu and T.-K. Leung, "Formal methods for protocol testing: A detailed study," *IEEE Transactions on Software Engineering*, vol. 15, no. 4, pp. 413–426, 1989.

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

[31] 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)*, ser. Springer Lecture Notes in Computer Science, vol. 3235. Madrid: Springer-Verlag, September 2004, pp. 115–132.

[32] 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.

[33] 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.

[34] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra, "Guided test generation for web applications," in *35th International Conference on Software Engineering (ICSE 2013)*. IEEE / ACM, 2013, pp. 162–171.

[35] C. D. Nguyen, A. Marchetto, and P. Tonella, "Combining model-based and combinatorial testing for effective test case generation," in *International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM, 2012, pp. 100–110.

[36] 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.

[37] H. Ural, X. Wu, and F. Zhang, "On minimizing the lengths of checking sequences," *IEEE Transactions on Computers*, vol. 46, no. 1, pp. 93–99, 1997.

[38] R. M. Hierons, "Minimizing the number of resets when testing from a finite state machine," *Information Processing Letters*, vol. 90, no. 6, pp. 287–292, 2004.

[39] R. M. Hierons and H. Ural, "Generating a checking sequence with a minimum number of reset transitions," *Automated Software Engineering*, vol. 17, no. 3, pp. 217–250, 2010.

[40] A. da Silva Simão and A. Petrenko, "Checking completeness of tests for finite state machines," *IEEE Transactions on Computers*, vol. 59, no. 8, pp. 1023–1032, 2010.

[41] M. Yao, A. Petrenko, and G. v. Bochmann, "Conformance testing of protocol machines without reset," in *Protocol Specification, Testing and Verification, XIII (C-16)*. Elsevier (North-Holland), 1993, pp. 241–256.

[42] D. Sidhu and T. K. Leung, "Experience with test generation for real protocols," in *ACM SIGCOMM 88*. ACM Press, 1988, pp. 257–261.

[43] B. Wang and D. Hutchison, "Protocol testing techniques," *Computer communications*, vol. 10, no. 2, pp. 79–87, 1987.

[44] G. Luo, A. Petrenko, and G. v. Bochmann, "Selecting test sequences for partially-specified nondeterministic finite state machines," in *The 7th IFIP Workshop on Protocol Test Systems*. Tokyo, Japan: Chapman and Hall, November 8–10 1994, pp. 95–110.

[45] R. Farber, *CUDA Application Design and Development*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

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

PLACE PHOTO HERE

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.

PLACE PHOTO HERE

Uraz Cengiz Türker received the BA, MSc and PhD degrees in Computer Science (Sabanci University, Turkey), in 2006, 2008, and 2014, respectively. He is now post doctoral researcher at Brunel University London under the supervision of Prof. Robert M. Hierons.