Parallel Algorithms for Generating Harmonised State Identifiers and Characterising Sets

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

Abstract—Many automated finite state machine (FSM) based test generation algorithms require that a characterising set (CS) or a set of harmonised state identifiers (HSIs) is first produced. The only previously published algorithms for partial FSMs were brute-force algorithms with exponential worst case time complexity. This paper presents polynomial time algorithms and also massively parallel implementations of both the polynomial time algorithms and the brute-force algorithms. In the experiments the parallel algorithms scaled better than the sequential algorithms and took much less time. Interestingly, while the parallel version of the polynomial time algorithm was fastest for most sizes of FSMs, the parallel version of the brute-force algorithm scaled better due to lower memory requirements.

Keywords—Software engineering/software/program verification, Software engineering/testing and debugging, Software engineering/test design, Finite State Machine, Characterising sets, Harmonised state identifiers, General purpose graphics processing units.

1 Introduction

Software testing is an important part of software development but typically is expensive, manual and error prone. One possible solution is to base testing on a formal model or specification [1], [2], allowing rigorous test generation algorithms to be used. In this context, one of the most widely used types of formal model is the Finite State Machine (FSM). The tester might devise an FSM to drive test generation and execution or an FSM might be produced from the semantics of a model in a more expressive language, such as Specification and Description Language (SDL) or State-Charts [3], [4], [5].

Approaches that derive a *test sequence* (an input sequence) from an FSM model have been developed in various application domains such as sequential circuits [6], lexical analysis [7], software design [8], communication protocols [5], [9], [10], [11], object-oriented systems [12], and web services [13], [14]. Such techniques have also been shown to be effective when used in important industrial projects [15]. Once a test sequence has been devised from an FSM specification M, the test sequence is applied to the implementation N. The output sequences produced by N and M are then compared and if they differ then we can deduce that N is faulty.

There are many techniques that automate the generation of test sequences from an FSM specification M, with this work going back to the seminal papers of Moore [16] and Hennie [17]. Most such techniques use *state identification sequences* that distinguish the states of M [8], [17], [18], [19], [20], [21], [22], [23]. For example, a test technique can check that the input of x in state s leads to output y and state s' as follows: start with a preamble (input sequence) $\bar{x}$ that takes M to s, then apply x, check that the output produced is y, and finally use one or more input sequences to distinguish the expected state s' from all other states of M. The test technique might also check the state reached by $\bar{x}$.

One approach to state identification is to use a characterising set (CS): a set of input sequences that distinguish all pairs of states [8], [24]. Harmonised state identifiers (HSIs) improve on CSs by allowing different sets of input sequences for different states [25]. One of the benefits of CSs and HSIs is that every minimal deterministic FSM[1] has a CS and an HSI and so test generation techniques that use these are generally applicable. This paper focuses on generating CSs and HSIs. The main focus of previous work has been complete FSMs, where the response to input x in state s is defined for every input x and state s. However, it has been observed that often FSM specifications are not complete (they are *partial*) [26], [27], [28], [29]. Complete FSMs are a special class of partial FSMs and so partial FSMs model a wider range of systems. However, the traditional state identification methodologies are not directly applicable to partial FSMs [17], [30]. The assumption that FSMs are complete is typically justified by assuming that missing transitions can be completed by, for example, adding transitions with null output.

Although it is sometimes possible to complete a partial FSM, this is not a general solution [31]. For example, an FSM M might specify a component that receives inputs from another component M'; the behaviour of M' influences what input sequences can be received by M. In addition, it might not be possible for certain input sequences to be provided by the environment due to physical constraints. Furthermore, ensuring completeness may introduce redundancy. For example, during the

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

1. Similar to most other work, we restrict attention to minimal deterministic FSMs; these terms are defined in Section 2.

experiments we found a benchmark FSM 'scf' provided in [32] that has 97 states and 13 million inputs but only 166 transitions. Clearly, it is not sensible to complete such an FSM. There has thus been interest in techniques for testing from a partial FSM [20], [21], [25], [31], [33], [34], [35], [36], [37], [38].

While one might expect a tester to use quite small models, an FSM M might represent the semantics of a model M' written in a more expressive language such as State-Charts or SDL. A state of M will represent a combinations of a logical state for M' and a tuple of values for the variables in M'. Thus, even small models can lead to large FSMs. However, the scalability of deriving CSs and HSIs from partial FSMs has received little attention despite FSM specifications often being partial and many FSM-based test generation methods using a CS or HSI. Note that by 'scalability' we refer to the size (number of states) of the largest partial FSMs that can be processed by an algorithm in a reasonable amount of time. To our knowledge there exists only one paper that proposes algorithms for deriving CSs and HSIs for partial FSMs [25] despite there being test generation algorithms, for testing from a partial FSM, that use such state identifiers [20], [21], [25], [31], [37], [38]. The CSI/HSI generation algorithms presented are sequential algorithms that operate on a single thread. The sequential CS algorithm has exponential worst case time complexity and the sequential HSI algorithm requires CSs to first be constructed.

Despite the increasing interest in *Graphics Processing Units (GPUs)* [39], [40], [41], [42], [43], [44], no previous work has utilised GPU technology to generate CSs or HSIs. In this work, our primary motivation is to address the scalability problem raised when constructing CSs and HSIs and to do so through using GPU technology.

As noted, the only published algorithm for generating CSs and HSIs from partial FSMs has exponential worst case time complexity. This paper tackles scalability from two directions. First, we devise a polynomial time sequential algorithm for generating CSs. We also produced a parallel implementation of this. We devise a parallel HSI construction algorithm and also a parallel implementation of the brute-force algorithms for generating CSs and HSIs (based on previous work). We also present the results of experiments. In experiments with randomly generated FSMs, the parallel algorithms scaled much better than the sequential algorithms. Such improvements in performance should help test generation techniques scale to larger FSMs. Interestingly, although the parallel brute-force (worst case exponential time) algorithms were slower than the parallel versions of the polynomial time algorithms, they scaled better. This is because the polynomial time algorithms have greater memory requirements. The parallel algorithms also outperformed the sequential algorithms on benchmark FSMs.

There were several challenges in designing a scalable massively parallel algorithm for deriving CSs and HSIs from partial FSMs. First, it was necessary to develop data structures that (1) can be processed quickly; (2) can be efficiently stored in GPU memory; and (3) can encapsulate enough data for constructing CSs and HSIs. It is also important that the parallel algorithm maximises GPU utilisation (occupancy). There is a trade-off between these factors since, for example, storing information in local GPU memory reduces memory access time but can also reduce the number of threads that can run in parallel.

This paper is organised as follows. We provide preliminary material in Section 2 and in Section 3 we develop a polynomial time sequential algorithm. In Section 4 we present the proposed parallel HSI and CS algorithms. Section 5 outlines the experiments and the results of these. Section 6 describes threats to validity and in Section 7 we draw conclusions. The appendix explains how the algorithms were implemented on a GPU.

2 PRELIMINARIES

2.1 Finite State Machines (FSMs)

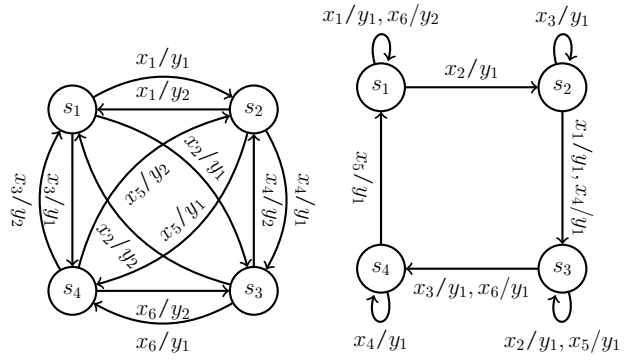
An FSM is defined by a tuple $M = (S, s_0, X, Y, h)$ where $S = \{s_1, s_2, \ldots, s_n\}$ is the finite set of states; $s_0 \in S$ is the initial state; $X = \{x_1, x_2, \ldots, x_p\}$ is the finite set of inputs; $Y = \{y_1, y_2, \ldots, y_q\}$ is the finite set of outputs (we assume that X is disjoint from Y); and $h \subseteq S \times X \times Y \times S$ is the set of transitions of M. We let $h = \{\tau_1, \tau_2, \ldots, \tau_\kappa\}$.

If $(s, x, y, s') \in h$ and input $x \in X$ is applied when M is in state s then M can change its state to s' and produce output y. Here $\tau = (s, x, y, s')$ is a *transition* of M with *starting state* s, *ending state* s', and *label* x/y. An FSM M is *deterministic* if for all $s \in S$ and $x \in X$ we have that M has at most one transition of the form (s, x, y, s').

An FSM can be represented by a directed graph. Figure 1 gives two FSMs with state sets $\{s_1, s_2, s_3, s_4\}$, inputs $\{x_1, x_2, x_3, x_4, x_5, x_6\}$, and outputs $\{y_1, y_2\}$. A node represents a state and a directed edge from a node labelled s to a node labelled s' with label x/y represents the transition $\tau = (s, x, y, s')$.

An input x is *defined at state* s if there exists a transition of the form $(s, x, y, s') \in h$ and we then use the notation $\delta(s, x) = s'$ and $\lambda(s, x) = y$. If x is not defined at state s then we write $\delta(s, x) = e$ (for a special symbol $e \notin S$) and $\lambda(s, x) = \varepsilon$. Thus δ is a function from $S \times X$ to $S \cup \{e\}$ and λ is a function from $S \times X$ to $Y \cup \{\varepsilon\}$. If for every state $s \in S$ and input $x \in X$, there exists a transition of the form (s, x, y, s') then M is a *complete FSM*, otherwise it is a *partial FSM*. In this paper, we consider partial deterministic FSMs and from now on we use the term FSM to refer to partial deterministic FSMs.

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. We use ε to represent the empty sequence and use $pref(.)$ to denote the set of all the prefixes of parameter $(.)$ longer than 0. For example, $pref(x_1x_2) = \{x_1, x_1x_2\}$ and $pref(\{x_1x_2, x_3x_4\}) = \{x_1, x_1x_2, x_3, x_3x_4\}$.



(a) FSM M_1 (b) FSM M_2

Figure 1: CS for M_1 has $4(4-1)/2$ elements $W = \{\{x_1\},\{x_2\},\{x_3\},\{x_4\},\{x_5\},\{x_6\}\}$. The length of separating sequence for states s_1, s_2 of M_2 is $4(4-1)/2$ and is $x_1x_2x_3x_4x_5x_6$.

The behaviour of an FSM M is defined in terms of the labels of walks of M. A *walk* is a sequence $\bar{\tau} = (s_1, x_1, y_1, s_2)(s_2, x_2, y_2, s_3)\dots(s_m, x_m, y_m, s_{m+1})$ of consecutive transitions. This walk has *starting state* s_1, *ending state* s_{m+1}, and *label* $x_1/y_1\ x_2/y_2 \dots x_m/y_m$. $\bar{z} = x_1/y_1\ x_2/y_2\ \dots\ x_m/y_m$ is an *input/output sequence*, $x_1x_2\dots x_m$ is the *input portion* ($in(\bar{z})$) of $\bar{z}$, and $y_1y_2\dots y_m$ is the *output portion* ($out(\bar{z})$) of $\bar{z}$. For example, M_1 has the walk $(s_2, x_4, y_1, s_3)(s_3, x_6, y_1, s_4)$ that has starting state s_2 and ending state s_4. The label of this walk is $x_4/y_1\, x_6/y_1$ and this has input portion x_4x_6 and output portion y_1y_1. FSM M is *strongly connected* if for every ordered pair (s, s') of states of M there is a walk with starting state s and ending state s'.

An input sequence $\bar{x} = x_1x_2\dots x_m$ is a *defined input sequence* for state s if M has a walk with starting state s and label $\bar{z}$ such that $in(\bar{z}) = x_1\dots x_m$. For example, since $(s_2, x_4, y_1, s_3)(s_3, x_6, y_1, s_4)$ is a walk of M_1, we have that x_4x_6 is a defined input sequence for s_2. We will let $L^I_M(s)$ denote the set of defined input sequences for s and $L^I_M(S') = \cap_{s\in S'} L^I_M(s)$ (the set of input sequences defined in all states in S'). In an abuse of notation we use λ and δ with input sequences: if x and $\bar{x}$ denote an input and an input sequence respectively such that $x\bar{x}$ is a defined input sequence for s then $\delta(s,\varepsilon) = s$, $\delta(s, x\bar{x}) = \delta(\delta(s,x),\bar{x}))$, $\lambda(s,\varepsilon) = \varepsilon$, $\lambda(s, x\bar{x}) = \lambda(s,x)\lambda(\delta(s,x),\bar{x})$. For example, in M_1 we have that $\delta(s_2, x_4x_6) = s_4$ and $\lambda(s_2, x_4x_6) = y_1y_1$. Note that at times we let $\bar{x}/\bar{y}$ denote the input/output sequence $\bar{z}$ such that $\bar{x} = in(\bar{z})$ and $\bar{y} = out(\bar{z})$. Given set S' of states and $\bar{x} \in L^I_M(S')$ we let $\delta(S', \bar{x}) = \cup_{s\in S'}\{\delta(s,\bar{x})\}$ and $\lambda(S', \bar{x}) = \cup_{s\in S'}\{\lambda(s,\bar{x})\}$.

We let $L_M(s)$ denote the set of labels of walks of M with starting state s and so $L_M(s) = \{\bar{x}/\bar{y} | \bar{x} \in L^I_M(s) \wedge \bar{y} = \lambda(s,\bar{x})\}$. For example, $L_{M_1}(s_2)$ contains the input/output sequence $x_4/y_1\ x_6/y_1$. FSM M defines the language $L(M) = L_M(s_0)$ of labels of walks with starting state s_0. Given $S' \subseteq S$ we let $L_M(S') = \cup_{s\in S'} L_M(s)$. States s, s' are *equivalent* if $L_M(s) = L_M(s')$ and FSMs M and N are *equivalent* if $L(M) = L(N)$.

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^* of length k. An input sequence $\bar{x} \in X^\star$ is a *separating sequence* for states s and s' if $\bar{x}$ is a defined input sequence for s and s' and $\lambda(s,\bar{x}) \neq \lambda(s',\bar{x})$. Consider, for example, M_1 (Figure 1a) and states s_1 and s_2. Then x_1 is a separating sequence for this pair since x_1 is defined in both states and $\lambda(s_1, x_1) = y_1 \neq y_2 = \lambda(s_2, x_1)$. In contrast, no input sequence starting with input x_4 can be a separating sequence for a pair that contains s_1 since x_4 is not defined in s_1. If every pair of states of FSM M has a separating sequence then M is a *minimal FSM*.

In this work, we consider only minimal FSMs. If an FSM is not minimal then a minimal FSM can be formed by merging pairs of compatible states in an iterative manner, where two states are compatible if they cannot be distinguished. It is possible to check whether two states are compatible in polynomial time and so it is also possible to construct a minimal FSM M' from a partial FSM M in polynomial time [45] [2]. The main restriction we make is that we consider deterministic FSMs. The main focus of FSM-based testing has been on deterministic FSMs and these have been used in areas such as hardware [47], protocol conformance testing [5], [8], [9], [11], object-oriented systems [12], web services [13], [48], [49], and general software [50].

There are a number of approaches to state verification and this paper focuses on two that are applicable to any minimal FSM. A characterisation set is a set of input sequences that, between them, distinguish all of the states of M.

Definition 2.1: A *CS* for FSM $M = (S, s_0, X, Y, h)$ is a set $W \subseteq X^\star$ such that for all $s_i, s_j \in S$ with $i \neq j$ there exists $\bar{x} \in W$ such that a prefix of $\bar{x}$ distinguishes s_i and s_j.

We use $\mathcal{S}_M = \{(s_i, s_j) | s_i, s_j \in S, i < j\}$ to denote a set of distinct pairs (s_i, s_j) of states of M. The restriction that $i < j$ ensures that a pair of states is represented exactly once. We will use $\mathcal{S}$ to denote $\mathcal{S}_M$ and γ_{ij} will denote a pair (s_i, s_j) in $\mathcal{S}_M$. Since $|S| = n$, set $\mathcal{S}$ contains $n(n-1)/2$ pairs. A CS W is a set of input sequences that distinguish the pairs in $\mathcal{S}$.

When considering a minimal *complete* FSM M with state set S ($|S| = n$), a set A of input sequences defines an equivalence relation $\sim_A$ over S, with two states s, s' being equivalent under $\sim_A$ if $\lambda(s,\bar{x}) = \lambda(s',\bar{x})$ for all $\bar{x} \in A$. Let us suppose that we add an input sequence $\bar{x}$ to A, to form A', and this makes A more effective at distinguishing the states of M: there exist $s, s' \in S$ such that $s \sim_A s'$ and $\lambda(s,\bar{x}) \neq \lambda(s',\bar{x})$. Then $\sim_{A'}$ has more equivalence classes than $\sim_A$. Since an equivalence relation $\sim_A$ on S can have at most n equivalence classes, it is thus straightforward to see that, starting from the empty set, we can add at most $n-1$ input sequence if each input sequence is to make the set more effective.

2. The problem of constructing a *smallest* such M' is NP-hard [46].

Thus, M has a CS with at most $n-1$ sequences. Further, we can use the following result, which is straightforward to prove, to deduce that there is such a CS where each input sequence is of length at most $n-1$.

Proposition 2.1: If input sequence $\bar{x}$ distinguishes states s, s', no proper prefix of $\bar{x}$ distinguishes these states, and $\bar{x} = \bar{x}'\bar{x}''$ for input sequences $\bar{x}'$ and $\bar{x}''$ then $\bar{x}''$ distinguishes states $\delta(s, \bar{x}')$ and $\delta(s', \bar{x}')$.

For minimal partial FSMs we have different bounds since a set A of input sequences need not define an equivalence relation on the states of a partial FSM M if different input sequences from A are defined from different states of M. In particular, it is possible to construct an FSM M such that for each pair of states s, s' there is a unique input x such that x is the only input defined in both s and s' (and so a characterising set must contain at least $n(n-1)/2$ input sequences). However, for a minimal partial FSM M we require at most one input sequence for each pair of states in $\mathcal{S}_M$ and so at most $n(n-1)/2$ sequences. We can use Proposition 2.1 to deduce that we require input sequences of length at most $n(n-1)/2$ and so a CS requires $O(n^4)$ memory. Figure 1 contains two FSMs. FSM M_1 has $n(n-1)/2$ inputs and for every input x_i there is a pair of states s, s' such that x_i is the only input that is defined in both s and s' and so a CS must contain an input sequence that starts with x_i. For FSM M_2 there is a pair of states whose shortest separating sequence is of length $n(n-1)/2$.

It may not be necessary to execute all sequences from a CS to distinguish a state s from all other states [25].

Definition 2.2: A *state identifier* (SI) for a state s_i of FSM $M = (S, s_0, X, Y, h)$ is a set $H_i \subseteq X^\star$ such that for all $s_j \in S \setminus \{s_i\}$, there exists $\bar{x} \in H_i$ such that $\bar{x}$ is a separating sequence for s_i and s_j.

This leads to HSIs.

Definition 2.3: A set of *Harmonised State Identifiers* (HSIs) for FSM $M = (S, s_0, X, Y, h)$ is a set of state identifiers $\mathcal{H} = \{H_1, H_2, \ldots H_n\}$ such that for all $s_i, s_j \in S$ with $i \neq j$, there exists $\bar{x} \in pref(H_i) \cap pref(H_j)$ that is a separating sequence for s_i and s_j.

As suggested in [25] one can derive HSIs from a CS. This provides an upper bound n^4 on the size of HSIs.

2.2 Previous HSI generation method

The HSI construction algorithm given in [25] takes an FSM M and CS W for M and in the first step the algorithm constructs an SI for the initial state (s_0). To do so the algorithm generates a subset H_0 of W such that for all $s_j \in S \setminus \{s_0\}$ there exists at least one input sequence $\bar{x}$ in H_0 such that $\bar{x}$ is a separating sequence for s_0 and s_j. The remaining SIs are computed in the second phase. For state s_i, the algorithm finds a subset H_i of W such that

1) For $j < i$, there exists $\bar{x} \in H_i$ and $\bar{x}' \in H_j$ such that some input sequence in $pref(\bar{x}) \cap pref(\bar{x}')$ distinguishes s_i and s_j.
2) For all s_j with $i < j$, there exists $\bar{x} \in H_i$ and a prefix $\bar{x}'$ of $\bar{x}$ with $\bar{x}' \in L^I_M(s_j)$ (the set of input sequences defined in s_j) such that $\bar{x}'$ distinguishes s_i and s_j.

3 Novel CS generation algorithm

The previously devised CS generation algorithm, for partial FSMs, has exponential worst case execution time. In this section we devise a polynomial time algorithm.

The first step of the proposed algorithm computes separating sequences of length one. The following is an immediate consequence of Proposition 2.1.

Proposition 3.1: Let M be a minimal FSM M. Then there exists a pair of states $(s, s') \in \mathcal{S}_M$ and an input $x \in X$ such that x distinguishes (s, s').

After the first step we have two sets of pairs of states: a set $P_\surd$ of pairs with separating sequences of length one and a set $P_\times$ of pairs whose separating sequences have not yet been computed. In the second step, the algorithm computes new separating sequences through using the previously computed separating sequences.

Proposition 3.2: Let M be a minimal FSM M, let $P_\surd \neq \emptyset$ be the set of pairs of states with known separating sequences, and let $P_\times = \mathcal{S}_M \setminus P_\surd$. Then there exists $(s, s') \in P_\times$, $(s'', s''') \in P_\surd$ with separating sequence $\bar{x}$, and a defined input $x \in X$ for s, s' such that $x\bar{x}$ is a separating sequence for (s, s').

We now present terminology used in this section. A *pair-node* η is a tuple (s_i, s_j, ϑ) such that $(s_i, s_j) \in \mathcal{S}_M$ $(i < j)$ and $\vartheta \in X^\star$ is a separating sequence for s_i and s_j or is ε if such a separating sequence has not yet been found. We use $\lambda(\eta, x)$ to denote $\{\lambda(s_i, x), \lambda(s_j, x)\}$. If $\vartheta = \varepsilon$, $\delta(\{s_i, s_j\}, x) = \{s, s'\}$ and there is a pair-node $\eta' = (s, s', \vartheta')$ with $\vartheta' \neq \varepsilon$ then we say that η 'evolves to' (becomes) $(s_i, s_j, x\vartheta')$. In such a situation, $x\vartheta'$ is a separating sequence for s_i, s_j.

The algorithm (Algorithm 1) initialises P; for each $(s, s') \in \mathcal{S}$ it adds pair-node $\eta = (s, s', \varepsilon)$ (Line 1). This requires $O(n^2)$ time. The algorithm then enters a loop (first-loop) in which it computes separating sequences of length one, iterating over P and X (Lines 2-4). The first-loop requires $O(n^2 p)$ time.

The algorithm then enters a while loop (main-loop) and computes separating sequences through evolving the elements of P. At each iteration the algorithm enters a for-loop that iterates over $P \times X$. For a pair-node η from set P with $\vartheta = \varepsilon$ and input $x \in X$, the algorithm checks whether η evolves to a pair-node η' (with x) that has separating sequence $\vartheta' \neq \varepsilon$. If so, the algorithm uses the separating sequence $\vartheta = x\vartheta'$ (Lines 6-8). If, after the for-loop, no pair-node has evolved, the algorithm declares that M is not minimal (Lines 9-10) and otherwise it continues. If all items in P have non-empty separating sequences, the algorithm returns P and terminates. To find a pair to be evolved the main loop can iterate $O(n^2 p)$ times; since there are $O(n^2)$ pairs, the main-loop requires $O(n^4 p)$ time. The main-loop is the most expensive component.

Theorem 3.1: If M is an FSM with n states and p inputs then Algorithm 1 requires $O(n^4p)$ time.

Algorithm 1: Sequential CS generation algorithm

Input: An FSM $M = (S, s_0, X, Y, h)$ where $|S| = n$ and $n > 1$
Output: CS for M if M is minimal
begin
1 $P \leftarrow \{(s_1, s_2, \varepsilon)(s_1, s_3, \varepsilon), \ldots (s_{n-1}, s_n, \varepsilon)\}$
// **first-loop**
2 **foreach** $\eta = (s_i, s_j, \vartheta) \in P, x \in X$ **do**
3 **if** $|\lambda(\eta, x)| > 1$ *and* $\varepsilon \notin \lambda(\eta, x)$ **then**
4 $\vartheta \leftarrow x$
// **main-loop**
5 **while** *There exist a pair* $\eta = (s_i, s_j, \vartheta) \in P$ *such that* $\vartheta = \varepsilon$ **do**
// **for-loop**
6 **foreach** $\eta = (s_i, s_j, \vartheta) \in P$ *such that* $\vartheta = \varepsilon$ *and* $x \in X$ *with* $\varepsilon \notin \lambda(\eta, x)$ **do**
7 **if** $\{s, s'\} = \delta(\{s_i, s_j\}, x)$, $\eta' = (s, s', \vartheta') \in P$ *and* $\vartheta' \neq \varepsilon$ **then**
8 $\vartheta \leftarrow x\vartheta'$
9 **if** *No pair-node is updated* **then**
10 Declare M is not minimal.
11 Return P

The following results, which demonstrate that Algorithm 1 is correct, follow immediately from the construction of the algorithm and Proposition 2.1.

Theorem 3.2: If Algorithm 1 is given a minimal FSM M then it returns a set P.

Theorem 3.3: If Algorithm 1 returns P when given minimal FSM M then P defines a CS for M.

4 Parallel CS and HSI algorithms

In this section we first provide the approach employed to address scalability problems and then the parallel CS and HSI generation algorithms.

4.1 Design Choices

There are two main strategies: the *Fat Thread* strategy and the *Thin Thread* strategy [51]. The fat thread approach minimises data access latency by having threads process a large amount of data on shared memory [51]. However, the number of threads may be restricted by the available shared memory and this may reduce performance.

In contrast, the thin thread approach aims to maximise the number of threads by storing only small amounts to data in shared memory. Although global memory transactions are relatively slow, it has been reported that the high global memory transaction latency can be hidden when there are many threads [51]. In this work we employed the thin thread strategy.

4.2 Parallel CS algorithm

4.2.1 Parallel CS algorithm: parallel design

To implement a thin thread based algorithm we propose what we call a *conditional pair-node vector* (CPn-vector for short), and later we will see that a scalable CS generation algorithm can be based on this. A CPn-vector $\mathcal{P}$ captures information related to pair-nodes plus additional information that will allow us to evolve its elements.

Definition 4.1: A conditional pair-node vector (CPn-vector) $\mathcal{P}$ for an FSM $M = (S, s_0, X, Y, h)$ with n states is a vector with $n(n-1)/2$ conditional pair-nodes. Each element ρ in the vector $\mathcal{P}$ is a tuple (f, η) for a *pair-node* $\eta = (s, s', \vartheta)$, and a *flag* $f : \{T, F\}$ that states whether states s, s' are distinguished by ϑ.

Let $\rho = (F, (s, s', \varepsilon))$ and $\rho' = (T, (s'', s''', \vartheta'))$ be CPns. As in the case of pair-nodes, we say ρ 'evolves' to ρ' with a defined input x, if $\delta(\{s, s'\}, x) = \{s'', s'''\}$. After the evolution, ρ becomes $(T, (s, s', x\vartheta'))$. As the elements of a CPn-vector evolve, we update the flag values.

The following are based on Theorem 3.3 and the definition of evolution of CPns and show how CPn-vectors are related to the states distinguished.

Lemma 4.1: Given CPn-vector $\mathcal{P}$, if $\mathcal{P}$ contains $\rho = (T, (s, s', \vartheta))$ then ϑ is a separating sequence for (s, s').

Theorem 4.1: Given FSM $M = (S, s_0, X, Y, h)$, if the flags of all elements of CPn-vector $\mathcal{P}$ of M are set to true then the input sequences retrieved from pair-nodes of $\mathcal{P}$ define a Characterising Set for M.

4.2.2 Parallel CS algorithm: an overview

Algorithm 2: Parallel CS generation algorithm, highlighted lines are executed in parallel.

Input: An FSM $M = (S, s_0, X, Y, h)$ where $|S| = n$ and $n > 1$
Output: CS for M if M is minimal
begin
1 $\mathcal{P} \leftarrow \{(F, (s_1, s_2, \varepsilon))(F, (s_1, s_3, \varepsilon)), \ldots (F, (s_{n-1}, s_n, \varepsilon))\}$
// **first-loop**
2 **foreach** $x \in X$ **do**
3 **if** *There exists* $\rho = (F, (s_i, s_j, \vartheta)) \in \mathcal{P}$ *such that* $\vartheta = \varepsilon$, $|\lambda(\eta, x)| > 1$ *and* $\varepsilon \notin \lambda(\eta, x)$ **then**
4 $\rho \leftarrow (T, (s_i, s_j, x))$
// **main-loop**
5 **while** *There exist an element* $\rho = (F, (s_i, s_j, \vartheta)) \in P$ **do**
6 **foreach** $\rho = (F, (s_i, s_j, \vartheta)) \in \mathcal{P}$ *and* $x \in X$ **do**
7 **if** $\{s, s'\} = \delta(\{s_i, s_j\}, x)$, $(T, (s, s', \vartheta')) \in P$ *and* $\vartheta' \neq \varepsilon$ **then**
8 $\rho \leftarrow (T, (s_i, s_j, x\vartheta'))$
9 **if** *no new conditional pair-node* ρ *is altered* **then**
10 Declare M is not minimal.
11 Return $\mathcal{P}$

The parallel-CS algorithm first initialises the CPn-vector $\mathcal{P}$ in parallel (Line 1). A naïve implementation would require $O(n(n-1)/2)$ time to initialise $\mathcal{P}$. However, as we will see later, the initialisation of $\mathcal{P}$ can be achieved in $O(n)$ time if $\Gamma \geq n$, where Γ is the number of threads used by the GPU.

Afterwards, the algorithm enters the first-loop in which it finds all elements that are distinguished by a single input (Lines 2-4). This step takes $O(n(n-1)p/(2\Gamma))$ time. The algorithm then enters the main-loop. In the main-loop the algorithm applies all inputs to elements of $\mathcal{P}$ whose flag is F and evolves elements. If an evolution to a distinguishes pair of states is possible then the flag values are set to T (Lines 6-8) (can be

achieved in $O(n(n-1)/(2\Gamma)p)$ time). If no elements of $\mathcal{P}$ have changed then the algorithm terminates, otherwise it continues (Lines 9-10). This step may also require $O(n(n-1)p/(2\Gamma))$ time. As the length of a separating sequence is bounded above by $n(n-1)/2$, the main-loop iterates $O(n^2)$ times and hence the algorithm requires $O(n^4p/\Gamma)$ time.

Theorem 4.2: If Algorithm 2 is given an FSM with n states and p inputs and there are Γ threads then it requires $O((n^4p)/\Gamma)$ time.

4.3 Parallel HSI algorithm

4.3.1 Parallel HSI algorithm: parallel design

The existing HSI generation algorithm takes as input a CS for the FSM M. However, we require $O(n^4)$ space to store a CS for an FSM with n states. This has at least two practical implications: (1) it may be impossible to keep data in the main memory and (2) threads will process a very large amount of data.

Recently, Hierons and Türker proposed a heuristic to construct HSIs for complete FSMs that overcomes this bottleneck [52]. Instead of using an existing CS, they construct HSIs from *incomplete distinguishing sequences*. Their algorithm keeps a list ($\mathcal{Q}$) of pairs of states (a list for the items of set $\mathcal{S}$) such that at each iteration an input sequence that removes the maximum number of pairs from $\mathcal{Q}$ is selected and the algorithm terminates when $\mathcal{Q}$ is empty. In the Parallel-HSI algorithm, we adopted this strategy by using *state-trace vectors*. A state-trace-vector contains the information regarding which pairs from $\mathcal{S}$ have been distinguished by an input sequence $\bar{x}$.

Definition 4.2: A state-trace vector (ST-vector) D for an FSM $M = (S, s_0, X, Y, h)$ with n states is a vector associated with input sequence $\bar{x} \in X^\star$ and having n elements such that: an element d of D is a tuple $(s_i, s_c, \bar{y})$ such that s_i is an *initial state*, $s_c = \delta(s_i, \bar{x})$ is a *current state*, and $\bar{y} = \lambda(s_i, \bar{x})$ is an output sequence.

We may need to construct a set of ST-vectors since there may not be a single input sequence that distinguishes all pairs of states.

Theorem 4.3: A set of state-trace vectors for an FSM M defines an HSI for M if for every pair of states s_i, s_j there exists a state-trace vector D associated with input sequence $\bar{x}$ such that there exists $\bar{x}' \in pref(\bar{x})$ that is a separating sequence for s_i and s_j.

4.3.2 Parallel-HSI algorithm: an overview

We now give a brief overview of the algorithm and in the Appendix we show how the Parallel-HSI algorithm was implemented using GPUs. For an FSM with n states, the parallel HSI algorithm uses a boolean valued vector $\mathcal{B}$ of length $n(n-1)/2$. The elements of $\mathcal{B}$ correspond to pairs in $\mathcal{S}$: the ith pair of $\mathcal{S}$ corresponds to the ith item of $\mathcal{B}$ and is set to 1 if these states have been distinguished. Initially all elements in $\mathcal{B}$ are set to 0 and when all elements are 1 the algorithm terminates.

The parallel HSI algorithm (Algorithm 3) has three nested loops. The first loop (*main-loop*) iterates as long as at least one element in $\mathcal{B}$ is 0. In every iteration, the algorithm checks whether the upper-bound on the length of the input sequence $\bar{x}$ has been reached. If so, the algorithm terminates. Otherwise, the algorithm enters the *middle-loop* and increments ℓ (initially $\ell = 0$). The middle-loop iterates as long as there exists an unprocessed input sequence of length ℓ and not all elements in $\mathcal{B}$ are 1.

In the middle-loop the algorithm first resets the ST-vector D. It then receives the next input sequence $\bar{x}$ of length ℓ and evolves elements in D with $\bar{x}$. Since the FSM is partially specified, the algorithm evolves an element if $\bar{x}$ is defined at the associated state. Then the algorithm executes the inner loop (*for-loop*). The for-loop iterates over the states and for state s_i it compares the output sequence obtained from s_i and all other states (in parallel). If there exists a state s_j that produces an output sequence that is different from that produced from s_i, then the algorithm writes 1 to the corresponding element of $\mathcal{B}$ (corresponding to γ_{ij} or γ_{ji}). Later it writes $\bar{x}$ to the corresponding SIs (H_is) of distinguished pairs. For state s_i there are $n-1$ pairs in $\mathcal{S}$ (and in $\mathcal{B}$) with state s_i and so the size of H_i cannot be larger than $n-1$.

The process of finding a separating sequence for a pair of states might require all possible input sequences to be considered. Therefore the worst case time complexity of the algorithm is exponential.

4.3.3 Generating characterising sets from HSIs

We first show how HSIs relate to characterising sets.

Lemma 4.2: Let $M = (S, s_0, X, Y, h)$ be an FSM and $\mathcal{H}$ be a set of harmonised state identifiers for M. Then $\bigcup_{H_i \in \mathcal{H}} H_i$ defines a characterising set of M.

Following the intuition provided by Lemma 4.2 we can construct a CS by using the Parallel-HSI algorithm through replacing Line 18 with the following line:

$$\textbf{18} \quad \text{return} \quad W = \bigcup_{H_i \in \mathcal{H}} H_i$$

It is possible to parallelise the process of taking the union of harmonised state identifiers through two steps. In the first step we sort all the input sequences in parallel and in the second step we pick unique state identifiers to form the CS. We present details in the Appendix.

5 Experiments

5.1 Experimental Design

The experiments had two main aims.

1) To explore how well the algorithms scaled. Therefore, we recorded the time taken.
2) To explore properties of the CSs and HSIs constructed. We recorded the number of sequences and the lengths of these sequences since fewer/shorter sequences lead to cheaper testing.

Initial experiments used FSMs generated by the tool used in [53]. This randomly assigns $\delta(s, x)$ and $\lambda(s, x)$

Algorithm 3: Parallel HSI construction algorithm, highlighted lines are executed in parallel

Input: An FSM $M = (S, s_0, X, Y, h)$ where $|S| = n$ and $n > 1$
Output: HSI for M
begin

1. $\mathcal{H} \leftarrow \{H_0 \leftarrow \emptyset, H_1 \leftarrow \emptyset, \ldots H_{n-1} \leftarrow \emptyset\}$
   // Construct vector D and a boolean vector $\mathcal{B}$ of size $|\mathcal{S}|$ such that the ith item of $\mathcal{B}$ corresponds to the ith pair of $\mathcal{S}$.
2. Construct D with n elements such that $\bar{x} \leftarrow \varepsilon$ and each element d_i is associated with initial/current state s_i and $\bar{y} \leftarrow \varepsilon$. Construct a boolean vector $\mathcal{B}$ of length $n(n-1)/2$ whose elements are 0.
3. $Execute \leftarrow T$, $\ell \leftarrow 0$
   // (***main-loop***)
4. **while** $Execute$ *is* T **do**
5. **if** $\ell + 1 \leq n(n-1)/2$ **then**
6. $\ell \leftarrow \ell + 1$
   else
7. Declare that FSM is not minimal and terminate
   // (***middle-loop***)
   while *There exists unprocessed* $\bar{x} \in X^\ell$ *and* $Execute$ *is* T **do**
   // For every element of D set current state value to initial state value
8. For every element d of D, $s_c \leftarrow s_i$
   // Replace $\bar{x}$ of D with a new input sequence $\bar{x}'$ retrieved from X^ℓ
9. $\bar{x} \leftarrow \bar{x}'$ for some $\bar{x}' \in X^\ell$ that has not yet been used
   // Evolve D with $\bar{x}'$
10. For every element d of D, $d \leftarrow evolve(d, \bar{x})$
    // Analyse the outcome of application of $\bar{x}$.
11. **foreach** $s_i \in S$ **do**
    // Mark distinguished states with $\bar{x}$ and store $\bar{x}$
12. **if** $\exists d_i, d_j \in D$ *with* $i \neq j$ *and* $\mathcal{B}[\gamma_{ij}] = 0$ *and* $\bar{x}' \in pref(\bar{x})$ *that is a separating sequence for* s_i, s_j **then**
13. $H_i \leftarrow H_i \cup \{\bar{x}\}$
14. $H_j \leftarrow H_j \cup \{\bar{x}\}$
15. Let $index$ denote the index of pair γ_{ij} on $\mathcal{S}$
    $\mathcal{B}[index] \leftarrow 1$
16. **if** *all elements in* $\mathcal{B}$ *are set to* 1 **then**
17. $Execute \leftarrow F$
18. Return $\mathcal{H}$.

for each $s \in S$ and $x \in X$, discarding the FSM if it is not strongly connected and minimal. After constructing M, the tool randomly selects $1 \leq K \leq np$ and K state-input pairs. For a pair (s, x) it erases the transition of M with start state s and input x. If deleting a transition τ disconnects M then τ is retained and another pair chosen. We used the tool to construct three test suites.

In test suite one ($TS1$), for each $n \in \{2^6, 2^7, \ldots 2^{17}\}$ we had 100 FSMs with number of inputs/outputs $p/q = 3/3$. These experiments explored the performance of the algorithms under varying numbers of states. To see the effect of the numbers of inputs and outputs we constructed $TS2$ and $TS3$. In $TS2$ we set $n = 1024$ and $q = 3$ and for each of $p \in \{2^4, 2^5 \ldots, 2^8\}$ we had 100 FSMs. For $TS3$ we set $n = 1024$ and $p = 3$ and for each of $q \in \{2^4, 2^5 \ldots, 2^8\}$ we again had 100 FSMs. Therefore we used 2200 FSMs[3].

It is possible that FSM specifications of real life systems differ from randomly generated FSMs. Therefore we also performed experiments on case studies: FSMs from the ACM/SIGDA benchmarks. This is a set of FSMs used in workshops in *1989–91–93* [32]. In Table 2 we present the specifications.

Name	$\lvert X\rvert$	$\lvert S\rvert$	$\lvert S\rvert * \lvert X\rvert$
$dk27$	2	7	14
$bbtas$	4	6	24
$dk17$	4	8	32
$dk15$	8	4	32
$s298$	8	135	1080
$ex7$	4	10	40
mc	5	15	75
$dk512$	2	15	30
$dk16$	4	27	108
$ex4$	64	14	21
tbk	64	16	776
$planet$	128	48	115
$s386$	128	13	1664
$bbsse$	128	13	1664
$s1$	256	18	4351
$ex1$	512	18	138
$styr$	512	30	166
$sand$	2048	32	184
scf	134217728	97	166

Figure 2: Properties of benchmark FSMs.

5.2 Experimental settings

Throughout this section we use SEQ-BF-CS to refer to the sequential brute force CS generation algorithm [25] and SEQ-BF-HSI to refer to the sequential brute force HSI generation algorithm [25]. PAR-BF-HSI, PAR-BF-CS, SEQ-PLY-CS, PAR-PLY-CS will denote the Parallel-HSI algorithm, the CS generation algorithm base on the Parallel-HSI algorithm, the sequential polynomial time CS generation algorithm, and the parallel polynomial time CS generation algorithm respectively. We set a bound of n on the length of sequences considered in the PAR-BF-CS algorithm. However, this did not affect the results regarding scalability since there were no cases where the PAR-BF-CS algorithm failed to find separating sequences and other algorithms returned separating sequences of length greater than n.

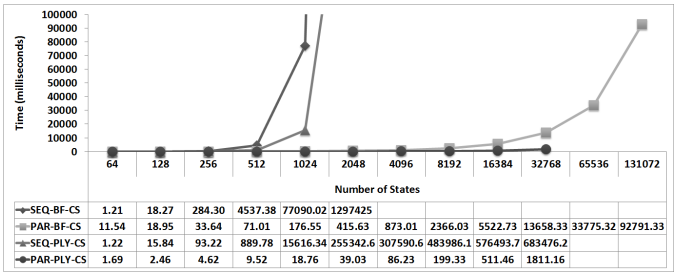
We used an Intel Core 2 Extreme CPU (Q6850) with 8GB RAM and NVIDIA TESLA K40 GPU under 64 bit Windows Server 2008 R2. During the experiments we stored the generated CSs and HSIs in the hard disk drive as for large FSMs the available CPU/GPU memory becomes insufficient. The timing information does not include the time for storing the sequences.

Finally, to perform the experiments in an acceptable amount of time, we set 1500 seconds as the limiting time.

5.3 The effect of the number of states

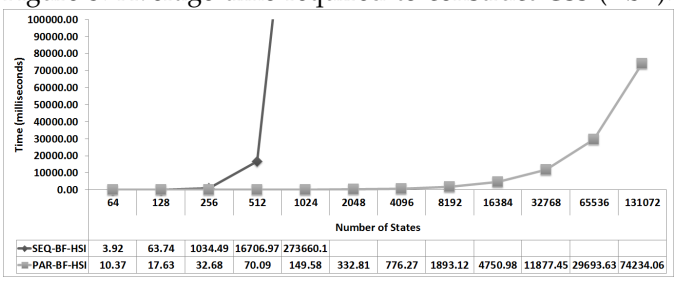
Figures 3 and 4 presents the mean construction times in ms. The results are promising: on average PAR-BF-HSI was 420 times faster than SEQ-BF-HSI and 1836 times faster when $n = 1024$. On average, PAR-BF-CS was 605 times faster than SEQ-BF-CS; SEQ-PLY-CS was 3 times

3. After an FSM is generated we do not check whether it is minimal. However, during experiments we found that 316 FSMs were not minimal. When such an FSM was found we simply generated a replacement and so each test suite contained 100 minimal FSMs.



	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072
SEQ-BF-CS	1.21	18.27	284.30	4537.38	77090.02	1297425						
PAR-BF-CS	11.54	18.95	33.64	71.01	176.55	415.63	873.01	2366.03	5522.73	13658.33	33775.32	92791.33
SEQ-PLY-CS	1.22	15.84	93.22	889.78	15616.34	255342.6	307590.6	483986.1	576493.7	683476.2		
PAR-PLY-CS	1.69	2.46	4.62	9.52	18.76	39.03	86.23	199.33	511.46	1811.16		

Figure 3: Average time required to construct CSs ($TS1$).



	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072
SEQ-BF-HSI	3.92	63.74	1034.49	16706.97	273660.1							
PAR-BF-HSI	10.37	17.63	32.68	70.09	149.58	332.81	776.27	1893.12	4750.98	11877.45	29693.63	74234.06

Figure 4: Average time required to construct HSIs ($TS1$).

faster than SEQ-BF-CS; and PAR-PLY-CS was 6316 times faster than SEQ-BF-CS. With $n = 2048$, PAR-BF-CS was 3118 times faster than SEQ-BF-CS and PAR-PLY-CS was 33241 times faster than SEQ-BF-CS.

The bottleneck for PAR-BF-HSI is the $\mathcal{B}$ vector, requiring $n(n-1)/2$ boolean variables. PAR-BF-HSI was able to process FSMs with 131072 states, making PAF-BF-HSI 128 times more scalable than the existing HSI construction algorithm. The bottleneck for PAR-PLY-CS is the $\mathcal{P}$ vector, requiring $2n(n(n-1)/2)$ integer values plus $n(n-1)/2$ boolean values. PAR-PLY-CS was able to process FSMs with 32768 states making PAR-PLY-CS 16 times more scalable than the sequential brute-force characterising set generation algorithm.

Table 1 shows the mean sequence lengths. For CSs there are no differences. In addition, the mean sequence length for PAR-BF-HSI was slghtly less than that for SEQ-BF-HSI. However, when we applied the parametric Kurskall Vallis [54] significance test[4], we found that the difference was not statistically significant.

Table 2 shows the mean size of state identifiers. PAR-BF-HSI tended to generate SIs that, on average, were slightly smaller than those returned by SEQ-BF-HSI. According to the Kruskal Vallis test there is a statistically significant difference when $n \geq 256$. This indicates that as the number of states increases, the parallel HSI generation algorithm tends to find fewer SIs. Moreover, we can see that the number of CSs generated does not depend on whether the algorithm is sequential or parallel. Overall, the results suggest that PAR-BF-HSI constructs more compact HSIs and faster and SEQ-PLY-CS and PAR-PLY-CS are faster than the brute-force versions.

4. The results obtained from SEQ-BF-HSI were the nominal and the results obtained from PAR-BF-HSI were the measurement variable

5.4 The effect of the number of inputs and outputs

The results for $TS2$ and $TS3$ are in Figure 5. As the number of inputs increases (Figures 5a, 5c), the time required to construct CSs and HSIs increases regardless of the algorithm used. However, mean lengths of state identifiers reduce as we increase the number of inputs (Figure 5e). This reflects there being more transitions that might be used in finding separating sequences. Figure 5g shows the mean number of separating sequences in HSIs, the differences between the HSIs constructed by SEQ-BF-HSI and PAR-BF-HSI being limited. The number of separating sequences constructed by SEQ-BF-CS, PAR-BF-CS, SEQ-PLY-CS and PAR-PLY-Cs are shown in Figure 5i. The number of separating sequences reduces with the number of inputs, indicating that when there are more inputs it was possible to find separating sequences that distinguish more states.

Consider now the experiments where we increased the number of outputs ($TS3$). The time required, the mean length of the separating sequences and mean number of separating sequences produced are in Figure 5b, Figure 5d, Figure 5f, Figure 5h, Figure 5j. The results are as expected: as the number of outputs increases it is easier to find separating sequences (one expects separating sequences to be shorter) and so the time taken reduces.

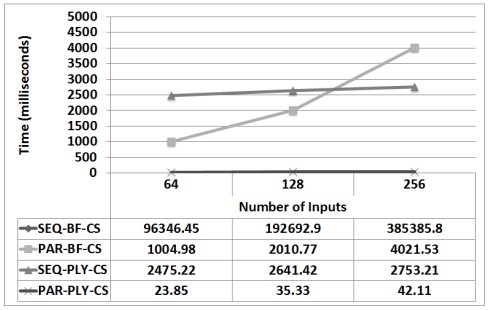
5.5 Benchmark FSMs

We found that PAR-BF-CS, PAR-PLY-CS, and SEQ-BF-CS generated identical CSs for each FSM except for scf. Only the PAR-BF-CS and PAR-PLY-CS algorithms were able to construct CSs for scf (Figure 6 gives the times). Moreover PAR-BF-HSI, and SEQ-BF-HSI generated identical state identifiers except for scf. SEQ-BF-HSI was not able to construct HSIs for scf.

The results are promising. Although the FSMs are relatively small, we see that the parallel algorithms outperformed the sequential algorithms.

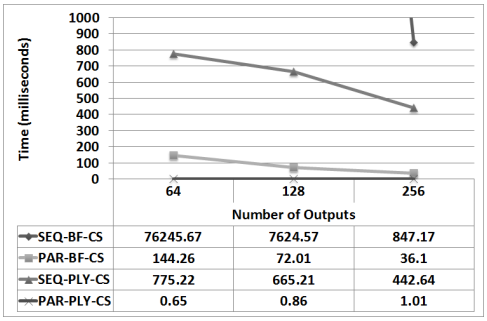
5.6 Discussion

The proposed algorithms accelerated the construction of CSs and HSIs and increased scalability. However, randomly generated FSMs will tend to have separating sequences that are much shorter than the upper bound. Sokolovskii introduced a special class of (complete) FSMs that we called *s-FSMs* [55]. The shortest separating sequence for states s_1 and s_2 of an s-FSM with n states has length $n-1$. We performed additional experiments with a set of s-FSMs to explore how the algorithms perform when the separating sequences are relatively long. The transition and the output functions of an s-FSM are defined as follows in which $n' = n/2$ and $n > 2$.



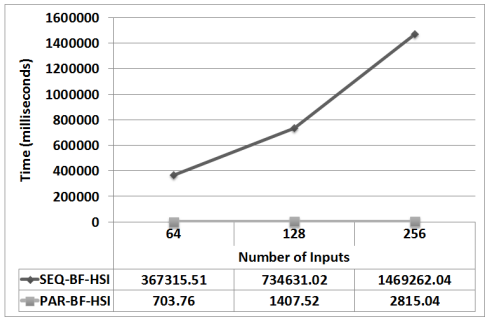
	64	128	256
SEQ-BF-CS	96346.45	192692.9	385385.8
PAR-BF-CS	1004.98	2010.77	4021.53
SEQ-PLY-CS	2475.22	2641.42	2753.21
PAR-PLY-CS	23.85	35.33	42.11

(a) Average time to construct CSs for FSMs in $TS2$.



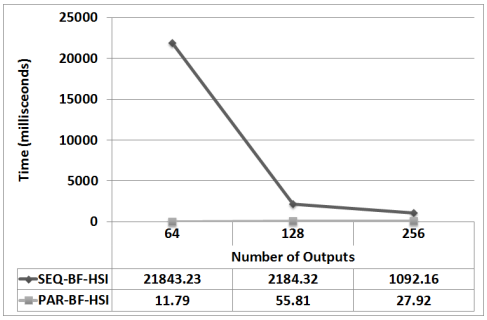
	64	128	256
SEQ-BF-CS	76245.67	7624.57	847.17
PAR-BF-CS	144.26	72.01	36.1
SEQ-PLY-CS	775.22	665.21	442.64
PAR-PLY-CS	0.65	0.86	1.01

(b) Average time to construct CSs for FSMs in $TS3$.



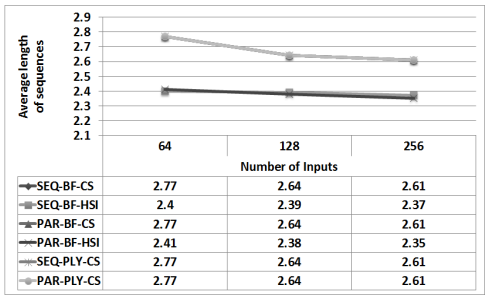
	64	128	256
SEQ-BF-HSI	367315.51	734631.02	1469262.04
PAR-BF-HSI	703.76	1407.52	2815.04

(c) Average time to construct HSIs for FSMs in $TS2$.



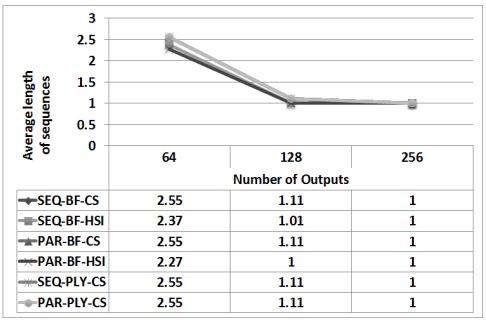
	64	128	256
SEQ-BF-HSI	21843.23	2184.32	1092.16
PAR-BF-HSI	11.79	55.81	27.92

(d) Average time to construct HSIs for FSMs in $TS3$.



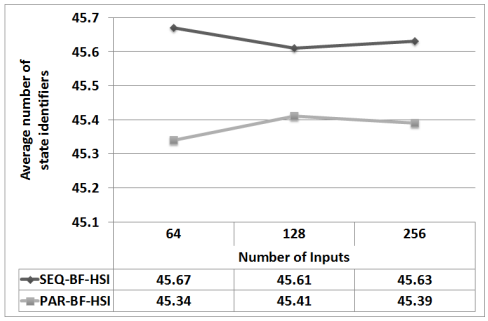
	64	128	256
SEQ-BF-CS	2.77	2.64	2.61
SEQ-BF-HSI	2.4	2.39	2.37
PAR-BF-CS	2.77	2.64	2.61
PAR-BF-HSI	2.41	2.38	2.35
SEQ-PLY-CS	2.77	2.64	2.61
PAR-PLY-CS	2.77	2.64	2.61

(e) Average length of sequences for FSMs in $TS2$.



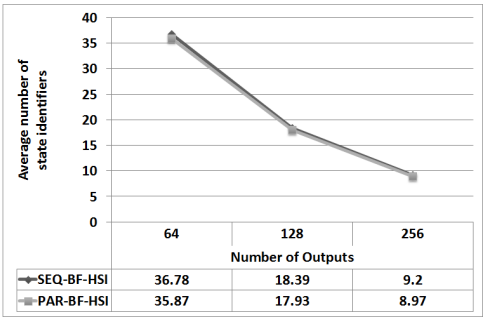
	64	128	256
SEQ-BF-CS	2.55	1.11	1
SEQ-BF-HSI	2.37	1.01	1
PAR-BF-CS	2.55	1.11	1
PAR-BF-HSI	2.27	1	1
SEQ-PLY-CS	2.55	1.11	1
PAR-PLY-CS	2.55	1.11	1

(f) Average length of sequences for FSMs in $TS3$.



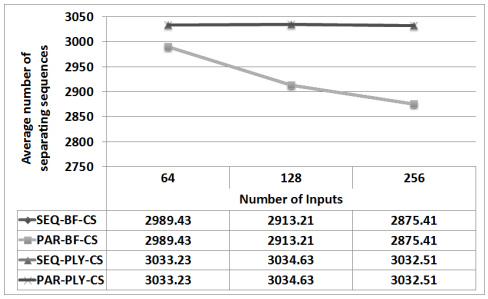
	64	128	256
SEQ-BF-HSI	45.67	45.61	45.63
PAR-BF-HSI	45.34	45.41	45.39

(g) Average number of state identifiers for FSMs in $TS2$.



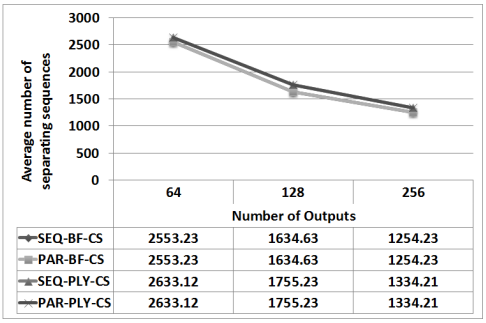
	64	128	256
SEQ-BF-HSI	36.78	18.39	9.2
PAR-BF-HSI	35.87	17.93	8.97

(h) Average number of state identifiers for FSMs in $TS3$.



	64	128	256
SEQ-BF-CS	2989.43	2913.21	2875.41
PAR-BF-CS	2989.43	2913.21	2875.41
SEQ-PLY-CS	3033.23	3034.63	3032.51
PAR-PLY-CS	3033.23	3034.63	3032.51

(i) Average number of separating sequences for FSMs in $TS2$.



	64	128	256
SEQ-BF-CS	2553.23	1634.63	1254.23
PAR-BF-CS	2553.23	1634.63	1254.23
SEQ-PLY-CS	2633.12	1755.23	1334.21
PAR-PLY-CS	2633.12	1755.23	1334.21

(j) Average number of separating sequences for FSMs in $TS3$.

Figure 5: Results of experiments on $TS2$ and $TS3$.

Algorithms	Number of States											
	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072
SEQ-BF-CS	3.21	3.22	3.34	3.76	3.88	4.23						
PAR-BF-CS	3.21	3.22	3.34	3.76	3.88	4.23	4.43	4.52	4.72	4.84	5.03	5.22
SEQ-PLY-CS	3.21	3.22	3.34	3.76	3.88	4.23	4.43	4.52	4.72	4.84		
PAR-PLY-CS	3.21	3.22	3.34	3.76	3.88	4.23	4.43	4.52	4.72	4.84		
SEQ-BF-HSI (per state)	3.05	3.11	3.19	3.22	3.45							
PAR-BF-HSI (per state)	3.05	3.09	3.15	3.19	3.46	3.68	3.86	4.25	4.69	4.96	5.13	5.37

Table 1: Average length of state identifiers generated for $TS1$.

Algorithms	Number of States											
	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072
SEQ-BF-CS	156.34	194.53	1200.56	1489.43	2935.45	7956.32						
PAR-BF-CS	156.34	194.53	1200.56	1489.43	2935.45	7956.32	13405.33	21230.35	58431.59	102440.91	173539.21	298552.78
SEQ-PLY-CS	190.54	233.11	1399.12	1611.55	3202.44	8301.22	15005.50	22532.07	68441.47	140448.02		
PAR-PLY-CS	190.54	233.11	1399.12	1611.55	3202.44	8301.22	15005.50	22532.07	68441.47	140448.02		
SEQ-BF-HSI (per state)	41.54	41.83	52.67	52.88	63.01							
PAR-BF-HSI (per state)	41.36	41.62	52.13	52.56	52.78	62.99	73.10	83.28	93.49	93.71	93.90	103.98

Table 2: Average number of state identifiers generated for $TS1$.

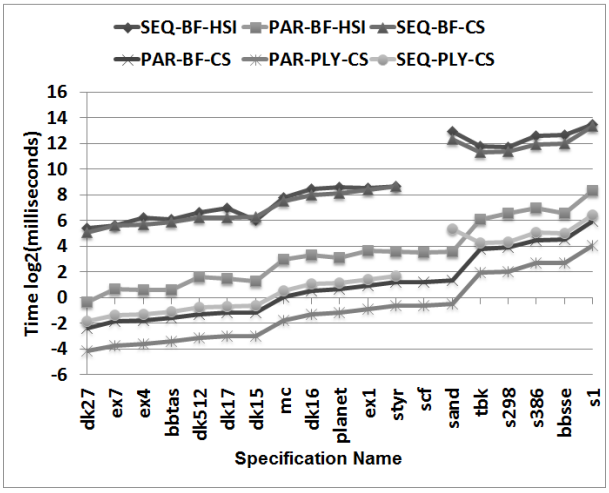


Figure 6: Results for benchmark FSMs.

	10	20	30
PAR-BF-HSI	2.56	1427.43	109746.34
SEQ-BF-HSI	0.01	1.32	2.21
PAR-BF-CS	2.51	1555.01	111000.34
SEQ-BF-CS	0.01	103.51	1553571.28

Table 3: The s-FSMs time in milliseconds.

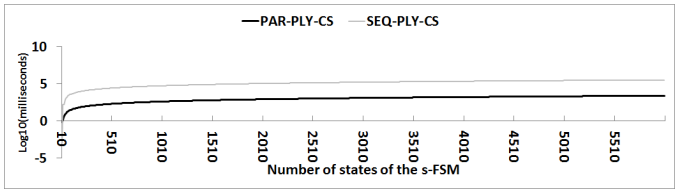


Figure 7: $\log_{10}$ scale of time (in ms) required to construct CSs for the s-FSMs with PAR-PLY-CS and SEQ-PLY-CS.

$$\delta(s_i, x_j) = \begin{cases} s_{i+1}, & \text{if } j = 1 \wedge i \neq n' \wedge i \neq n \\ s_1, & \text{if } j = 1 \wedge i = n' \\ s_{n'+1}, & \text{if } j = 1 \wedge i = n \\ s_i, & \text{if } j = 0 \wedge 1 \leq i \leq n' - 1 \\ s_{n'+1}, & \text{if } j = 0 \wedge i = n' \\ s_{i-n'}, & \text{if } j = 0 \wedge n' + 1 \leq i \leq n \end{cases} \quad (1)$$

$$\lambda(s_i, x_j) = \begin{cases} y_0, & \text{if } j = 0 \wedge i = n \\ y_1, & \text{otherwise} \end{cases} \quad (2)$$

We generated 600 s-FSMs with n states, $n \in \{10, 20, \ldots, 6000\}$. The results for the brute-force approaches (Table 3) indicate that scalability drops drastically for FSMs with long separating sequences. SEQ-BF-CS, PAR-BF-HSI, and PAR-BF-CS can process s-FSMs with up to 30 states. While SEQ-BF-HSI is very fast, this is because it does not construct separating sequences: for state s it picks state identifiers from CSs that have previously been constructed. However, it requires the output of SEQ-BF-CS and so SEQ-BF-HSI also is not scalable for such FSMs. However, SEQ-PLY-CS and PAR-PLY-CS were able to process all 600 s-FSMs. When $n = 30$, SEQ-PLY-CS required 150 milliseconds and PARL-PLY-CS required 2.98 milliseconds and when $n = 6000$ SEQ-PLY-CS required 307 seconds and PARL-PLY-CS required 2.2 seconds. We present the results of these experiments in Figure 7. These suggest that for s-FSMs, SEQ-PLY-CS and PAR-PLY-CS are at least 200 times more scalable than SEQ-BF-CS, PAR-PLY-CS is 537000 times faster than SEQ-BF-CS and SEQ-PLY-CS is 10357 times faster than SEQ-BF-CS.

We also investigated the effect of K by examining the relationship between performance and K for PAR-BF-HSI when $n = 4096$ in $TS1$. In Table 4 we observe that as we increase the number of transitions we increase the number of separating sequences but reduce the mean separating sequence length and the time required by PAR-BF-HSI. These results unsurprising since there being many transitions typically allows pairs of states to be distinguished using shorter sequences.

6 Threats to Validity

Threats to internal validity concern factors that might introduce bias and so largely concern the tools used in the experiments. The tool that generated the FSMs, used as experimental subjects, is one that has previously

Properties	Number of Transitions						
	5000	6000	7000	8000	9000	10000	11000
Number of separating sequences per state	9.85	21.45	38.45	75.45	90.45	104.45	110.45
Average length of separating sequences per state	6.84	6.13	4.79	2.94	2.57	1.84	1.73
Average to time construct separating sequences	904.45	882.35	819.44	737.67	679.11	649.45	648.63

Table 4: The effect of number of transitions on the distinguishability of FSMs where $n = 4096$, $p/q = 3/3$.

been used and we also tested this. We carefully checked and tested the implementations of the algorithms. We used C++ to code the sequential CS and HSI methods and CUDA C++ to implement the parallel algorithms. We used the CUDA-Thrust library for sorting output sequences while constructing characterising sets.

Threats to construct validity refer to the possibility that the properties measured are not those of interest in practice. Our main concern was scalability and this is important if FSM based test techniques are to be applied to larger systems. However, the sets of separators will typically be generated to be used within a test generation algorithm and so we are also interested in the potential effect on the size of the resultant test. There are two aspects that affect this: the number of sequences in a CS/HSI and the lengths of these sequences. We therefore measured mean values for these in the experiments.

Threats to external validity concern the degree to which we can generalise from the results. One cannot avoid such threats since the set of 'real FSMs' is not known and we cannot uniformly sample from this. To reduce this threat we varied the number of states, inputs and outputs. We also used FSMs from industry.

7 Conclusions and Future work

This paper explored the use of GPU computing to devise massively parallel algorithms for generating CSs/HSIs. The main motivation was to make such algorithms scale to larger FSMs. An FSM M used in test generation could represent the semantics of a model M' written in a more expressive language such as State-Charts or SDL and even quite modest models can result in large FSMs.

We used the thin thread strategy in which relatively little data is stored in shared memory, this allowing there to be many threads. We devised polynomial time algorithms and massively parallel versions of the brute-force and polynomial time algorithms.

We performed experiments with randomly generated partial FSMs, the parallel algorithms being much quicker than the sequential algorithms and scaling much better. Interestingly, the parallel brute-force algorithm, with exponential worst case complexity, outperformed the sequential polynomial time algorithm. The parallel version of the polynomial time algorithm was fastest but did not scale as well as the parallel brute-force algorithm due to its memory requirements. When the techniques were applied to FSMs with relatively long separating sequences, only the polynomial time algorithms scaled to FSMs with 40 states or more. However, these algorithms scaled quite well, easily handling FSMs with 6000 states. As expected, the parallel version of the polynomial time algorithm was much faster than the sequential version.

There are several lines of future work. First, experiments might explore how the results change when using different GPU cards. It would also be interesting to run additional experiments with FSMs generated from models in languages such as SDL and State-Charts. Third, there is the challenge of designing and implementing parallel algorithms for CS and HSI algorithms that can run on multi-core systems. Finally, there is the potential to investigate new approaches that are capable of constructing HSIs for larger FSMs.

8 Acknowledgements

This work was supported by The Scientific and Technological Research Council of Turkey (TUBITAK) under grant 1059B191400424 and by the NVIDIA corporation.

References

[1] M. Broy, B. Jonsson, and J.-P. Katoen, *Model-Based Testing of Reactive Systems: Advanced Lectures LNCS*. Springer, 2005.

[2] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. A. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Computing Surveys*, vol. 41, no. 2, pp. 9:1–9:76, 2009.

[3] A. Y. Duale and M. U. Uyar, "A method enabling feasible conformance test sequence generation for EFSM models," *IEEE Transactions on Computers*, vol. 53, no. 5, pp. 614–627, 2004.

[4] W. Grieskamp, "Multi-paradigmatic model-based testing," in *Formal Approaches to Software Testing and Runtime Verification FATES/RV*, ser. LNCS, vol. 4262. Springer, 2006, pp. 1–19.

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

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

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

[9] 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.

[10] S. Low, "Probabilistic conformance testing of protocols with unobservable transitions," in *1993 International Conference on Network Protocols*, Oct, pp. 368–375.

[11] 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.

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

[13] M. Haydar, A. Petrenko, and H. Sahraoui, "Formal verification of web applications modeled by communicating automata," in *Formal Techniques for Networked and Distributed Systems FORTE*, ser. LNCS, vol. 3235. Madrid: Springer-Verlag, September 2004, pp. 115–132.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] H. Ural and K. Zhu, "Optimal length test sequence generation using distinguishing sequences," *IEEE/ACM Transactions on Networking*, vol. 1, no. 3, pp. 358–371, 1993.

[20] G. L. Luo, G. v. Bochmann, and A. Petrenko, "Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method," *IEEE Transactions on Software Engineering*, vol. 20, no. 2, pp. 149–161, 1994.

[21] A. Petrenko, N. Yevtushenko, and G. v. Bochmann, "Testing deterministic implementations from nondeterministic FSM specifications," in *IFIP TC6 9th International Workshop on Testing of Communicating Systems*. Darmstadt, Germany: Chapman and Hall, 9–11 September 1996, pp. 125–141.

[22] A. Petrenko and A. Simão, "Generalizing the DS-methods for testing non-deterministic fsms," *The Computer Journal*, vol. 58, no. 7, pp. 1656–1672, 2015.

[23] A. da Silva Simão, A. Petrenko, and N. Yevtushenko, "On reducing test length for FSMs with extra states," *Software Testing, Verification and Reliability*, vol. 22, no. 6, pp. 435–454, 2012.

[24] M. P. Vasilevskii, *Failure Diagnosis of Automata. Cybernetics*. Plenum Publishing Corporation, 1973.

[25] 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.

[26] P.-C. Tsai, S.-J. Wang, and F.-M. Chang, "FSM-based programmable memory BIST with macro command," in *2005 IEEE International Workshop on Memory Technology, Design, and Testing, 2005. (MTDT)*, Aug., pp. 72–77.

[27] K. Zarrineh and S. Upadhyaya, "Programmable memory BIST and a new synthesis framework," in *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, June, pp. 352–355.

[28] L. Xie, J. Wei, and G. Zhu, "An improved FSM-based method for BGP protocol conformance testing," in *International Conference on Communications, Circuits and Systems*, 2008, pp. 557–561.

[29] A. Drumea and C. Popescu, "Finite state machines and their applications in software for industrial control," in *27th Int. Spring Seminar on Electronics Technology: Meeting the Challenges of Electronics Technology Progress*, vol. 1, 2004, pp. 25–29.

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

[31] ——, "Testing from partial deterministic FSM specifications," *IEEE Transactions on Computers*, vol. 54, no. 9, pp. 1154–1165, 2005.

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

[33] N. Kushik, N. Yevtushenko, and A. R. Cavalli, "On testing against partial non-observable specifications," in *9th International Conference on the Quality of Information and Communications Technology (QUATIC 2014)*. IEEE, 2014, pp. 230–233.

[34] A. L. Bonifácio and A. V. Moura, "Test suite completeness and partial models," in *12th International Conference on Software Engineering and Formal Methods (SEFM 2014)*, ser. LNCS, vol. 8702. Springer, 2014, pp. 96–110.

[35] ——, "Partial models and weak equivalence," in *11th International Colloquium on Theoretical Aspects of Computing ICTAC 2014)*, ser. LNCS, vol. 8687. Springer, 2014, pp. 80–96.

[36] ——, "On the completeness of test suites," in *Symposium on Applied Computing (SAC 2014)*. ACM, 2014, pp. 1287–1292.

[37] A. Petrenko and N. Yevtushenko, "Conformance tests as checking experiments for partial nondeterministic FSM," in *5th Int. Workshop on Formal Approaches to Software Testing*, ser. LNCS, vol. 3997. Springer, 2006, pp. 118–133.

[38] A. da Silva Simão and A. Petrenko, "Generating checking sequences for partial reduced finite state machines," in *20th IFIP TC 6/WG 6.1 International Conference Testing of Software and Communicating Systems, 8th International Workshop on Formal Approaches to Testing of Software TestCom/FATES*, ser. LNCS, vol. 5047. Springer, 2008, pp. 153–168.

[39] J. Dümmler and S. Egerland, "Interval-based performance modeling for the all-pairs-shortest-path problem on GPUs," *The Journal of Supercomputing*, vol. 71, no. 11, pp. 4192–4214, 2015. [Online]. Available: http://dx.doi.org/10.1007/s11227-015-1514-9

[40] F. Busato and N. Bombieri, "An efficient implementation of the bellman-ford algorithm for kepler GPU architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2015.

[41] H. Liu and H. H. Huang, "Enterprise: Breadth-first graph traversal on GPUs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 68:1–68:12.

[42] F. Al Farid, M. Uddin, S. Barman, A. Ghods, S. Das, and M. Hasan, "A novel approach toward parallel implementation of BFS algorithm using graphic processor unit," in *2015 International Conference on Electrical Engineering and Information Communication Technology (ICEEICT)*, May 2015, pp. 1–4.

[43] S. Lai, G. Lai, G. Shen, J. Jin, and X. Lin, "Gpregel: A GPU-based parallel graph processing model," in *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS)*, Aug 2015, pp. 254–259.

[44] H. Djidjev, G. Chapuis, R. Andonov, S. Thulasidasan, and D. Lavenier, "All-pairs shortest path algorithms for planar graph for GPU-accelerated clusters," *Journal of Parallel and Distributed Computing*, vol. 85, pp. 91 – 103, 2015, {IPDPS} 2014 Selected Papers on Numerical and Combinatorial Algorithms.

[45] G. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955.

[46] C. P. Pfleeger, "State reduction in incompletely specified finite-state machines," *IEEE Transactions on Computers*, vol. 22, no. 12, pp. 1099–1102, Dec. 1973.

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

[48] 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.

[49] 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)*. IEEE / ACM, 2013, pp. 162–171.

[50] 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)*. ACM, 2012, pp. 100–110.

[51] G. Klingbeil, R. Erban, M. Giles, and P. Maini, "Fat versus thin threading approach on GPUs : Application to stochastic simulation of chemical reactions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 2, pp. 280–287, Feb 2012.

[52] R. M. Hierons and U. C. Türker, "Incomplete distinguishing sequences for finite state machines," *The Computer Journal*, vol. 58, no. 11, pp. 3089–3113, 2015.

[53] ——, "Distinguishing sequences for partially specified FSMs," in *6th NASA Formal Methods Symposium (NFM)*, 2014, pp. 62–76.

[54] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American Statistical Association*, vol. 47, no. 260, pp. pp. 583–621, 1952.

[55] M. Sokolovskii, "Diagnostic experiments with automata," *Kibernetica*, no. 6, pp. 44–49, 1971.

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

[57] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010, version 1.7.0. [Online]. Available: http://thrust.github.io/

Symbol	Meaning
t_i	Thread with id i
$CurrentStates$	A vector of states
$InputSequence$	A vector of input symbols
$OutputSequence$	A vector of output symbols
FSM	A vector holding FSM transition structure
$Flags$	A vector of boolean variables

Table 5: Nomenclature for the algorithms.

APPENDIX

We first discuss issues affecting performance and then describe how the parallel algorithms were implemented. Table 5 gives the terms used.

Performance considerations

In implementing a massively parallel algorithm, one needs to consider *coalesced memory transaction* and *thread divergence*. As we followed the thin thread strategy, we needed to perform many global memory transactions. In a GPU, global memory is accessed in chunks of aligned 32, 64 or 128 bytes. If the threads of a block access global memory in the right pattern, the GPU can pack several accesses into one memory transaction. This is a *coalesced memory transaction* [56]. For example, whenever a thread t_i requests a single item, the entire line from global memory is brought to cache. If thread t_{i+1} requests the neighbouring item, t_{i+1} can read this data from the cache. As reading global memory is hundreds of times slower than reading cache memory, coalesced memory access may drastically improve performance. Therefore, to reduce the time spent on global memory transactions, we needed an appropriate storage layout.

All threads in a multiprocessor execute the same code (kernel). If the code has branching statements such as *if* or *switch* then some of the threads may follow different branches and hence different threads need to execute different lines of code. In such cases the GPU will serialise execution: a GPU is not capable of executing *if* or *switch* like statements in parallel. This problem is known as *thread divergence* [56]. However, if one can guarantee that the threads execute the same sequence of instructions then one can use branching statements.

Parallel-CS algorithm

Data structures

In implementing the Parallel-CS algorithm we used the $\mathcal{P}$ vector and the FSM vector. The $\mathcal{P}$ vector holds $n(n-1)/2$ CPn-vectors and we use $\mathcal{P}[i]$ to denote the ith element of $\mathcal{P}$. Such an element holds a pair-node and a flag where a node-pair consists of a pair of states (s, s') and possibly a separating sequence ϑ for this pair. Although the maximum length of ϑ is $n(n-1)/2$, we set this bound as n, and therefore the memory required for a CPn-vector was of $O(n^3)$.

The transition structure of the FSM is kept in the FSM vector. For state s and input x, the FSM vector returns output $y \in Y \cup \{\varepsilon\}$ and next state $s' \in S \cup \{e\}$. The size of the FSM vector is therefore $2n|X|$. For a thread t_i and an input sequence $\bar{x}$ of length greater than 1, reads on the FSM vector may not be coalesced as the memory access pattern on FSM is data dependent. For example, let us assume that we need to apply $\bar{x} = x_1x_2$ to s_i. For x_1, thread t_i will retrieve output and next state (s_j) information from the ith location of FSM and it will then apply input x_2 to s_j which will cause thread t_i to access a different location of the FSM vector.

Initiating the $\mathcal{P}$ vector

First note that for a pair (s_i, s_j) there are $n-i$ elements in $\mathcal{P}$ that start with s_i. Therefore, if an initialisation kernel receives the $\mathcal{P}$ vector, integers i and n as its parameters and launches with $n-i$ threads, it can set $n-i$ elements of $\mathcal{P}$. Since i varies from 1 to $n-1$, the Host can call the kernel $O(n)$ times. If $\Gamma \geq n$ the time required for initialisation is of $O(n)$ as stated in Section 4.2.

Computing initial separating sequences

To compute the initial separating sequences, we can launch a kernel with $n(n-1)/2$ threads and with the FSM and the $\mathcal{P}$ vectors, inputs X and a boolean variable $isMin$ (set to F prior to the kernel call) as parameters. Thread t_i processes one element $\mathcal{P}[i] = ((s, s', \vartheta), f)$ in a for-loop. The for-loop iterates over inputs X and in each iteration the kernel applies input x to states s, s' and retrieves outputs from the FSM vector and then checks if $\lambda(s, x) \neq \lambda(s', x)$, $\lambda(s, x) \neq \varepsilon$, and $\lambda(s', x) \neq \varepsilon$. If these conditions are met, the thread sets the flag to 1, separating sequence to x, and $isMin$ to T and exits from the for-loop. More than one thread can set $isMin$; as the only value to be written to $isMin$ is T, this does not cause a race. After all threads have finished, the kernel returns and the algorithm checks the value of $isMin$. If it is F, the algorithm declares that M is not minimal and the algorithm terminates, otherwise it continues.

Evolving the $\mathcal{P}$ vector

After some separating sequences are set, the algorithm uses these to compute additional separating sequences. To achieve this we implemented two kernels.

The first kernel is launched with $n(n-1)/2$ threads, and the FSM vector, the $\mathcal{P}$ vector, n, and inputs X as parameters. A thread t_i processes $\mathcal{P}[i] = (f, (s, s', \vartheta))$ through a for-loop if and only if $f = 0$. The for-loop iterates over inputs; at each iteration it applies an input to s and s' and receives $s_i = \delta(s, x)$ and $s_j = \delta(s', x)$ from the FSM vector. t_i then computes the index k for pair (s_i, s_j). If the flag of $\mathcal{P}[k]$ is 1 then the thread retrieves the separating sequence ϑ', assigns $x\vartheta'$ to ϑ, and sets $f = 2$. The thread sets the flag value to 2 rather than 1 to avoid long separating sequences being constructed. For example, consider the scenario in which thread t_i tries to evolve $\mathcal{P}[i]$ and there is a freshly evolved element $\mathcal{P}[j]$ (evolved in the current iteration) and an element $\mathcal{P}[!]$ that

has already evolved. As t_i checks input symbols one by one, if the flags of $\mathcal{P}[j]$ and $\mathcal{P}[!]$ are 1 then t_i may use ϑ_j in forming its separating sequence. However, $|\vartheta_j| > |\vartheta_!|$. By setting the flags of freshly evolved elements to 2, we ensure that these cannot be used to form new separating sequences until the next iteration.

After the first kernel returns, the algorithm launches another kernel with $\mathcal{P}$ and $isMin$ to convert 2s to 1s. A thread t_i checks if the flag of $\mathcal{P}[i]$ is 2, if so it sets the flag to 1 and sets $isMin$ to T. After the kernel returns, the algorithm checks $isMin$ and if it reads F it terminates declaring that M is not minimal; otherwise it continues.

Parallel-HSI algorithm

Data structures

The Parallel-HSI algorithm uses a boolean vector $\mathcal{B}$, of size $n(n-1)/2$, to record which pairs of states have known separating sequences. Recall that elements of an ST-vector are associated with an input sequence, initial and current states and output sequences. We simulated an ST-vector by using $CurrentStates$, $InputSequence$ and $OutputSequence$ vectors. The $InputSequence$ vector holds the input sequence $\bar{x}$ to be applied and so has length at most $n(n-1)/2$. $CurrentStates[i]$ gives the current state $\delta(s_i, \bar{x})$ for initial state s_i and so $CurrentStates$ has size n. The $OutputSequences$ vector holds the output sequences produced from the different states and so its length can vary from n to $n(n(n-1)/2)$.

The Parallel-HSI algorithm uses the FSM vector. The algorithm also uses a vector $Flags$: for state s, $Flags[i]$ is larger than 0 if and only if s_i has been distinguished from s. As the FSM and $InputSequence$ vectors never change they are kept in the Texture memory.

Resetting/Initiating Vectors

The Parallel-HSI algorithm first initialises the $\mathcal{B}$ vector using a kernel in which thread t_i writes 0 to $\mathcal{B}[i]$. The algorithm then enters a loop and in every iteration it resets vector D. As D is simulated by different vectors, we used more than one kernel to achieve this. The first kernel receives $CurrentStates$ and its length and during execution thread t_i writes s_i to $CurrentStates[i]$. The second kernel receives a vector ($InputSequence$, $Flags$, $OutputSequences$ etc.) and its size. During execution a thread t_i writes 0 to the ith index of the vector.

Evolving the ST vector

A thread t_i applies the input sequence $\bar{x}$ to state s_i, iterating over a loop (kernel-loop). The number of iterations of the kernel-loop is equal to the length of the input sequence. At each iteration of the kernel-loop, a thread t_i reads the next input x from the $InputSequences$ vector and retrieves the next state s and the observed output y from the FSM vector. It writes s to $CurrentState[j]$ and writes the output observed in the current iteration to the corresponding index of the $OutputSequence$ vector.

Evaluating the output sequences

After the elements of D have been evolved, the Parallel-HSI algorithm evaluates the output sequences through a loop (states-loop) called by the CPU. At each iteration, the states-loop chooses a state s_i and calls a kernel that writes 0 to all elements of the $Flags$ vector. It then executes another kernel in which a thread t_j compares the output sequences produced by states s_i and s_j through a loop (kernel-loop-2). At each iteration of kernel-loop-2, thread t_j retrieves outputs y_i, y_j respectively and checks whether the value $\varepsilon \oplus y_i \wedge \varepsilon \oplus y_j$ is 0 (here $\oplus$ denotes XOR). If so the thread is suspended since the input sequence is not a defined sequence (from at least one of s_i and s_j). Otherwise it writes $y_i \oplus y_j \vee Flags[j]$ to $Flags[j]$.

Gathering state identifiers

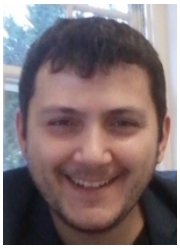
After the output sequences have been compared, a kernel compares the values of $Flags$ and $\mathcal{B}$. An input sequence is added to H_i and H_j if it distinguishes s_i and s_j ($Flags[j]$ is true) and the states were not previously distinguished ($!\mathcal{B}[index]$ is true). Thus, t_j writes $InputSequence$ to H_j, H_i and 1 to $\mathcal{B}[index]$ if $Flags[j] \wedge !\mathcal{B}[index]$ is larger than 0.

Constructing a characterising set from HSIs

While constructing the CS W from HSIs we need to eliminate duplicates. Thus, once HSIs have been computed by the Parallel-HSI algorithm we form a group $\mathcal{G}$ by collecting state identifiers from every H_i. We then sort these input sequences ($sort(\mathcal{G})$) in parallel, removing duplicated using the $generateW(\mathcal{G})$ kernel. We used the Thrust Sort function [57] to sort input sequences.

The $generateW$ kernel receives $\mathcal{G}$ and an empty W. A loop iterates $|\bar{x}_i|$ times and at each iteration a thread t_i compares the jth input of neighbouring input sequences: it compares the jth input of $\bar{x}_i$ with both $\bar{x}_{i+1}$ and $\bar{x}_{i-1}$. If t_i finds that $\bar{x}_i$ is different from a neighbouring input sequence then it adds this input sequence to W.

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.



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.