Efficient state synchronisation in model-based testing through reinforcement learning

1st Uraz Cengiz Türker
The School of Informatics
The University of Leicester
Leicester, UK
u.c.turker@leicester.ac.uk

2nd Robert M. Hierons
Department of Computer Science
The University of Sheffield
Sheffield, UK
r.hierons@sheffield.ac.uk

3rd Mohammad Reza Mousavi
Department of Informatics
King's College London
London, UK
mohammad.mousavi@kcl.ac.uk

4th Ivan Y. Tyukin
Department of Mathematics
The University of Leicester
Leicester, UK
i.tyukin@leicester.ac.uk

***Abstract*—Model-based testing is a structured method to test complex systems. Scaling up model-based testing to large systems requires improving the efficiency of various steps involved in test-case generation and more importantly, in test-execution. One of the most costly steps of model-based testing is to bring the system to a known state, best achieved through synchronising sequences. A synchronising sequence is an input sequence that brings a given system to a predetermined state regardless of system's initial state. Depending on the structure, the system might be complete, i.e., all inputs are applicable at every state of the system. However, some systems are partial and in this case not all inputs are usable at every state. Derivation of synchronising sequences from complete or partial systems is a challenging task. In this paper, we introduce a novel Q-learning algorithm that can derive synchronising sequences from systems with complete or partial structures. The proposed algorithm is faster and can process larger systems than the fastest sequential algorithm that derives synchronising sequences from complete systems. Moreover, the proposed method is also faster and can process larger systems than the most recent massively parallel algorithm that derives synchronising sequences from partial systems. Furthermore, the proposed algorithm generates shorter synchronising sequences.**

***Index Terms*—Model based testing, synchronising sequence, reinforcement learning, Q-learning**

I. Introduction

Model-based testing [1]–[6] is a rigorous method for validation and verification. The application of model-based testing to areas such as autonomous reactive systems has been further facilitated in recent years by advances in model-learning techniques, and their extensions to adaptive and evolving systems [7], [8]. The industrial-scale application of model-based testing in these challenging areas requires the scaling up of various parts of the underlying techniques.

Many systems are state-based: they have an internal state that affects behaviour and is updated by events and operations. When testing a state-based system, one typically needs to apply a number of test sequences, or adaptive test cases, from the initial state of the system under test (SUT). As a result, the state of the SUT is normally reset to this initial state between test sequences. Sometimes it is straightforward to reset the state of the SUT by, for example, turning the system off and then on again or issuing the reset signal provided by the manufacturer/developer. However, the process of resetting the SUT between test-cases can be one of the most time-consuming parts of test execution [9] and there may be no simple approach to resetting the SUT; there may be no reset input or the use of a reset input may be infeasible. In such circumstances, the reset operation is implemented through the use of *synchronising sequences* (SSs) [10]. SSs are used to bring the underlying system to a specific state to resume testing with a new test case [2], [9], [11]–[13]. In such cases, the length of SSs affects the cost of test execution. This motivates the work in this paper, which utilises reinforcement learning in the generation of short SSs. The development of efficient techniques that produce short SSs will contribute to the agenda of producing systematic MBT techniques that scale to large models.

In addition to model-based testing, synchronising sequences are also heavily used in motion planing of robotic controllers such as orienting parts on assembly lines [14]. Moreover, SSs have also been studied in automata theory and genomics and form an active area of research [12], [14]–[27].

A. Problem statement & Related Work

The underlying transition function of a reactive system test model can be *complete* or *partial*. In a complete model, all inputs of the system are applicable in each of its states. The development of efficient methods to generate short SSs from such a model is very appealing as the use of shorter SSs reduces the cost and time of test execution. However, the problem of generating a minimal SS, from a complete model, is **NP**-*hard* [26].

Even though there are several methods for generating an SS [26], [28]–[35], the initial step of all these methods is the construction of a *product-automaton*; a data structure that requires quadratic space with respect to the number of states of the model. All existing methods use this data-structure as the basis of different approaches/heuristics to derive short SSs. Amongst these, the fastest known algorithm [26] requires $O(n^3 + kn^2)$ time where n denotes the number of states and k denotes the number of inputs of the model.

To address the computational cost, recent work [36] introduced many-core and multi-core approaches for deriving short synchronising sequences from complete systems. However, it is relatively difficult to implement these methods, as they require (i) proficiency in parallel programming and programming general purpose graphics processing units (GPGPUs) and (ii) sophisticated graphics cards and CPUs.

In a partial model, not all inputs are applicable in all of the states of the model. For such models, the problem of deciding whether there is an SS is **PSPACE**-*complete* [37]. As a result, the current focus of research is merely related to the derivation of SSs. In this line of research, the first published algorithm was based on constructing a breadth first search tree, which requires exponential space and time and hence is not efficient [38], [39]. The only known efficient algorithm [15] checks all input sequences that are not longer than some upper-bound. Experimental results show that a combination of a relatively powerful GPU card and CPU can process systems with $16,000,000$ states and 10 inputs. However, the performance of the algorithm degrades as the number of inputs increases. Moreover, this method also requires (i) solid expertise in programming general purpose graphics processing units and (ii) a powerful graphics card.

Finally, in [40], the authors introduce an algorithm that uses *Answer Set Programming* to generate SSs from complete systems. As the method requires *a Conjunctive Normal Form* formulation of the system and the SS generation problem, it is not scaleable to large systems with many states.

Given recent breakthroughs in the Reinforcement Learning (RL) spectra, the objective of the work presented in this paper is to use RL as the basis for novel algorithms that efficiently find short synchronisation sequences. RL defines a family of machine learning algorithms and is becoming a major tool in computational intelligence [41]. In RL, computers (*agents*) make their own choices (take *actions*) in a given environment without having prior information or labelled data [42].

Recently, the use of RL in test generation has received significant attention [43]–[51] and we now briefly review a few examples. RL has formed the basis of an algorithm to automatically generate test cases for Android applications with the aim of improving code coverage [50] and has also been used in mutation testing to predict whether a mutant will be killed by a given test suite without incurring the cost of executing the mutant [52]. RL has also been used in graphical user interface (GUI) testing [44], [45].

In online testing, test inputs are derived during test execution. It has been shown that RL can be used to address the associated problem of optimizing the choice of test actions to reduce test costs [51]. RL has been used to learn a behaviour model of the system under test to aid risk-based testing [47]. Three RL algorithms have been proposed and embedded in EvoSuiteFIT [43] to support hyperheuristic search based test generation algorithms [48].

RL has also been used in security testing. For example, researchers have developed an RL based testing algorithm that trains dishonest agents to reveal dangerous behaviours of autonomous cyber-physical systems [49]. An RL based test generation technique has also been devised with the aim of increasing hardware Trojan detection accuracy [53], [54].

This paper uses Q-learning, which defines a family of RL methods in which the actions taken by an agent are decided according to quality values (Q-values) [55]. Q-learning has become the basis of many RL algorithms because unlike other methods, Q-learning is simple and exhibits excellent learning facilities [56]. Q-learning techniques have recently attracted significant attention with the introduction of Deep Q Networks (DQNs) [57]. Although Q-learning algorithms are promising, it is still challenging to use them in our setting. The initial Q-learning algorithm has good convergence properties, however it requires tables or matrices to hold Q-values. When the agent interacts with the environment, a single Q-value on the table is updated. So this simple setting introduces at least two problems, (i) in circumstances where the environment is complicated, critical states might not be experienced/learned (generalisation problem), and (ii) because the underlying data structure is a table, it leads to a large amount of storage space being required. Therefore, for complex learning problems with large, complex environments, it is difficult to achieve effective learning by using the tabular Q learning algorithm. Function approximators have been used, to overcome generalisation problem, as an alternative to keeping Q values [58].

B. Research questions

In light of these findings, the research questions studied in this paper are summarised as follows:

RQ 1 Is our proposed Q-learning algorithm more efficient and scalable than the state of the art algorithms?
We further refine this research question into the following sub-questions pertaining to the execution time, scalability, and memory usage, respectively:

- RQ 1.1 Is the growth of the execution time more modest in the Q-learning algorithm?
- RQ 1.2 How scalable can a Q-learning algorithm be? By fixing the amount of memory and execution time, can a Q-learning algorithm generate SSs from specifications that cannot be processed by the state of the art SS generation algorithms?
- RQ 1.3 How does the memory usage behaviour of the Q-learning algorithm compare to state of the art SS generation algorithms?

RQ 2 Is our proposed Q-learning algorithm more effective in generating shorter synchronising sequences for a larger class of system models?
We also refine this research question into the following sub-questions pertaining to the length of the generated synchronising sequence and the types of systems handled by the algorithm, respectively:

- RQ 2.1 Does the Q-learning algorithm generate shorter SSs (relative to execution time and memory consumption) than the state of the art SS generation algorithms?
- RQ 2.2 Can the Q-learning algorithm be extended to mitigate the above-mentioned generalisation problem?

C. Contributions

In this paper, we report what is, to the best our knowledge, the first application of reinforcement learning to deriving state synchronisation sequences, in order to make model-based testing more efficient. In particular, we propose a sequential, $\mathcal{Q}$-learning (reinforcement learning) algorithm that derives synchronising sequences from partial and complete systems.

Regarding deriving SSs using machine learning, apart from the work surveyed before, we are aware of only one closely related publication [59]. In this work, the authors report a Deep Learning method to predict the length of the SS without generating it. We, however, aim to generate the SS from a given specification.

Regarding our novel $\mathcal{Q}$-graph formalism, we are aware of two related pieces of work that used different types of data structures in the $\mathcal{Q}$-learning setting to ease the generalisation problem. In one approach [60], the Markov Decision Process (MDP) search space has been represented as a k dimensional tree which makes it possible to keep unrelated part of the environment in a database. This reduces the memory cost of the RL algorithm. A second piece of work [61] shows how the steps taken by an agent can be represented as a graph. This graph is then used to aid exploration/exploitation dilemma using a shortest path algorithm. None of these methods, however, employ a similar technique to ease the generalisation problem as the presented method.

Our $\mathcal{Q}$-learning framework, unlike the tabular setting, does not require prior knowledge of the entire search space. Instead, it explores and constructs the search space as the agent interacts with the environment. Therefore with this framework, we are allowed to apply the $\mathcal{Q}$-learning method to problems that possess a very large search space using a limited memory space. For example, the classical tabular $\mathcal{Q}$-learning method would require 2^n rows and in total $x * 2^n$ cells for a given system with n states and x inputs.

Through empirical evaluation, we show that the proposed algorithm is superior to the state of the art sequential and parallel algorithms: the proposed algorithm can generate shorter SSs and 1000 (256) times faster, and can process 256 (2048) times larger specifications than the state of the art sequential (parallel) algorithm. To ensure the scalability of our approach, we retrieved the specification of an industrial-scale system to manage the engine status of Océ printers and copiers (a subsidiary of Canon). The specification has 3410 states and 77 inputs; it has been developed in an industrial context and has been formally cross-checked against the design documents of the system, from which the implementation code is automatically generated [62]. Our method could derive an SS from this specification in less than a second where other state of the art SS generation methods would not terminate in an hour.

Our proposed algorithm serves as a basis for future efficient algorithms for large state-space exploration that cannot be explored exhaustively.

D. Organisation of the paper

The paper is organised as follows. In the next section, we provide the terminology and notation regarding reactive systems and $\mathcal{Q}$-learning used throughout the paper. This is then followed by a section in which we explain the proposed algorithm. In Section IV, we present the experimental subjects, conducted experiments, and their results. Later, in Section V, we discuss the results. Finally, Section VI draws conclusions and discusses some future research directions.

II. Preliminaries

A. Automata

We use the standard notation $A = \langle S, \Sigma, H \rangle$ to denote an automaton. Here S is a set of states, Σ is a finite input alphabet and $H \subseteq S \times \Sigma \times S$ is the set of transitions. The set of all input sequences is represented by $\Sigma^\star$. We let $|S'|$ denote the number of elements in a given set of states.

We let *dom-A* denote the set of pairs $(s, x) \in S \times \Sigma$ such that there exists a transition leaving state s with input x and we refer it as '*x is defined in state s*'. A is said to be *completely-specified* if *dom-A* $= S \times \Sigma$ and otherwise A is *partial*. A transition of A is represented by tuple $\tau = (s, x, s') \in H$, where s is the starting state, s' is the ending state, and x is the input label of the transition τ.

An input x is said to be *defined* at state s if and only if $(s, x) \in$ *dom-A*. Otherwise x is *undefined* at state s.

A *walk* of an automaton A is a sequence $(s_1, x_1, s_2)(s_2, x_2, s_3) \ldots (s_i, x_i, s_j)(s_j, x_j, s_k)$ of consecutive transitions of A. The input sequence of this walk $\omega = x_1 x_2 \ldots x_i x_j$ is called its *trace*, we use ε to denote the empty sequence. We can extend the notion of a defined input to defined input sequences as follows. An input sequence ω is *defined* at a given set of states $S' \subseteq S$ if one of the following is true (i) ω is an empty sequence; or (ii) for any prefix ω_1 of ω, with $\omega = \omega_1\omega_2$, ω_1 is defined at S' and ω_2 is defined at the states reached from S' using ω_1.

Let $\delta : 2^S \times \Sigma^\star \rightarrow 2^S$ be a map specifying the set of states which can be reached using an input sequence from a given set of states. Let $\omega = x\omega'$ be an input sequence. If for a given $s \in S'$, ω is not defined, then $\delta(S', \omega)$ is also not defined. Otherwise the delta function can be defined in a recursive way $\delta(S', \varepsilon) = S'$, $\delta(S', x) = \cup_{s \in S'} \delta(s, x)$, and $\delta(S', \omega) = \delta(\delta(S', x), \omega')$. If for some pair of states $s, s' \in S'$, $\delta(s, \omega) = \delta(s', \omega)$ then ω is said to be a *Merging Input Sequence* (MIS). If $\forall s_i, s_j \in S, \delta(s_i, \omega) = \delta(s_j, \omega)$ then ω is called a *Synchronising Sequence* (SS).

In Figure 1[1], we provided the specification for the ceiling speed monitoring with service brake intervention (SBI) from [63]. Note that this specifications is partial. This can be completed by adding self-loop transitions that label missing input, and this completion method is well known in this field [64]. Note that an input sequence $SS_1 = ?c3?c5?c6?c7$ resets the automaton to state $Normal$, i.e., $\delta(S, ?c3?c5?c6?c7) = \{Normal\}$.

While testing a given system, we need to apply a number of test sequences, all of which start from the reset state ($Normal$ in this case), and there is a need to reset the system under test between the execution of these test sequences. Thus, the length of the SS has an impact on test execution time and shorter SSs are preferable.

[1]We did not draw the added transitions in Figure 1.

The automaton A_{SBI} given in Figure 1 has another SS $SS_2 = ?c3?c0?c7$ that also resets the system to state $Normal$. Since SS_2 is 25% shorter, using SS_2 is preferable for testing. For example, consider the classical test generation method, the W method [2], [13]. Asymptotically, the W method can generate a test suite with $|S| + |S| * |\Sigma|$ elements. If we use SS_2 instead of SS_1 then we would save $(|S| + |S| * |\Sigma|) = 40$ inputs during testing an implementation of automaton A_{SBI}.

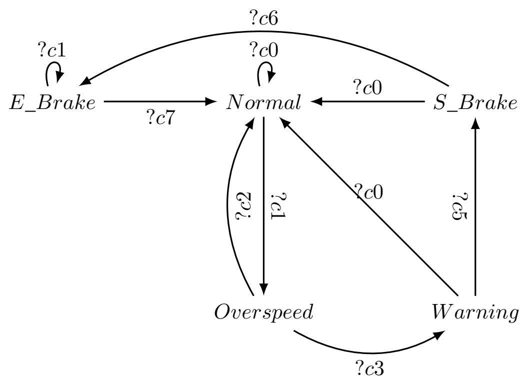


Fig. 1: Automaton A_{SBI} with $S = \{Normal, Warning, Overspeed, S\_Brake, E\_Brake\}$, $\Sigma = \{?c0, ?c1, ?c2, ?c3, ?c5, ?c6, ?c7\}$.

B. $\mathcal{Q}$-learning environment

$\mathcal{Q}$-learning algorithms can operate on a Markov Decision Process (MDP) [56]. An MDP is defined with a tuple $P = (\mathbb{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where $\mathbb{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $\mathcal{P}$ is the probability function in the form of $\mathcal{P}(\mathcal{S}'|\mathcal{S}, a)$, that is for each action $a \in \mathcal{A}$ and state $\mathcal{S} \in \mathbb{S}$, it defines the probability of reaching state $\mathcal{S}' \in \mathbb{S}$ and $\mathcal{R}$ is the immediate reward function in the form of $\mathcal{R}(\mathcal{S}, a, \mathcal{S}')$, i.e., it returns the reward received after transitioning from state $\mathcal{S}$ to $\mathcal{S}'$ with action a.

The $\mathcal{Q}$-learning is a value-based reinforcement learning method which is used to find the optimal policy when state transition probabilities $\mathcal{P}$ are *unknown* for a given MDP $P = (\mathbb{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$. Instead of estimating these unknown probabilities, the method uses a *value function*, $\mathcal{Q}$ [55]. The value function $\mathcal{Q}$ is recursively defined as

$$\mathcal{Q}(\mathcal{S}, a) = \begin{cases} \mathcal{Q}(\mathcal{S}, a) - \alpha\{R(\mathcal{S}, a) + \gamma * \arg\max_{a'} \mathcal{Q}(\mathcal{S}', a') \\ \quad -\mathcal{Q}(\mathcal{S}, a)\}, \text{ if } \mathcal{S} = \text{current state, and} \\ \quad\quad \text{an action } a \text{ is executed} \\ \\ \text{no change,} \quad \text{otherwise,} \end{cases} \tag{1}$$

where $\mathcal{S} \in \mathbb{S}$, $a \in \mathcal{A}$, and $\mathcal{Q}(\mathcal{S}, a)$ is the value of applying action a at state $\mathcal{S}$, $R(\mathcal{S}, a)$ is the immediate reward received after applying action a at state $\mathcal{S}$, α is the learning rate, and γ is the future reward discount factor. α, and γ are values within the range $[0, 1]$.

The $\mathcal{Q}$-learning algorithm asymptotically reconstructs the true expected discounted reward [55] and as a result works towards recovering the optimal policy. In this respect, policy selection based on $\mathcal{Q}$-learning can be viewed as an off-policy *temporal difference control* algorithm which asymptotically approximates the optimal policy [56].

III. The proposed algorithm

A. From $\mathcal{Q}$-tables to $\mathcal{Q}$-graphs and problem formulation

The naive implementation of the $\mathcal{Q}$-learning algorithm relies on a $\mathcal{Q}$-table which holds the $\mathcal{Q}$-values for each of the states of the underlying environment i.e., MDP P [56]. This is a drastic improvement over cases where complete knowledge of the transition probabilities $\mathcal{P}$ is required. The use of a $\mathcal{Q}$-table is straightforward: if the agent wants to learn the $\mathcal{Q}$-value of a state $\mathcal{S} \in \mathbb{S}$ it just reads the information from the table after it computes its index.

However, consider a scenario in which the agent does not require the whole $\mathcal{Q}$-table while learning the environment. In such a case, we may not have to store the $\mathcal{Q}$-values using a preset table but instead we can use a directed graph. Using such an approach, we let the graph grow as the agent discovers states on-the-fly. This method will reduce the memory requirements of the algorithm in situations in which the agent can find an optimum policy based on only a portion of the state-space, which is the case in our application domain.

A $\mathcal{Q}$-graph has a finite set of nodes N ($\mathcal{Q}$-*nodes*), such that each node $n \in N$ is associated with a state of the environment (say MDP state) $\mathcal{S}$ ($n_{\mathcal{S}}$) and a $\mathcal{Q}$-value ($\mathcal{Q}(\mathcal{S}, a)$) for each action ($a$) of P. Intuitively, in the set of admissible edges in a $\mathcal{Q}$-graph there exists an edge labelled with action a from $\mathcal{Q}$-node n to $\mathcal{Q}$-node n' if and only if one can reach $n_{\mathcal{S}'}$ from $n_{\mathcal{S}}$ using an action a.

Finally, we represent the SS construction problem as an MDP P. An element S' of the power set of S ($S' \in pow(S)$) corresponds to a state ($\mathcal{S}' \in \mathbb{S}$) of the MDP P and each input $x \in X$ of A is an action $a \in \mathcal{A}$ of P.

We use one-to-one and onto functions to denote corresponding inputs and set of states: (i) $i()$ maps an input $x \in \Sigma$ of the automaton A to the corresponding action $a \in \mathcal{A}$ of the MDP P and vice versa, and (ii) $st()$ maps a set of states $S' \in \mathcal{P}(S)$ of the automaton A to the corresponding state $\mathcal{S}' \in \mathbb{S}$ of the MDP P and vice versa.

For each $\mathcal{S}$ and a, we let

$$\mathcal{P}(\mathcal{S}'|\mathcal{S}, a) = \begin{cases} 1, \text{ if and only if } \delta(st(\mathcal{S}), i(a)) = st(\mathcal{S}'), \\ 0, \text{ otherwise.} \end{cases}$$

That is in P there exists a transition from a given state $\mathcal{S}$ to another state $\mathcal{S}'$ labelled with action a if and only if $\delta(st(\mathcal{S}), i(a)) = st(\mathcal{S}')$.

The immediate rewards ($R(\mathcal{S}, a)$) are computed as follows, let $\mathcal{S}'$ be the next MDP state such that $\delta(st(\mathcal{S}), i(a)) = st(\mathcal{S}')$.

$$R(\mathcal{S}, a) = \begin{cases} -1, & \text{if } |st(\mathcal{S})| = |st(\mathcal{S}')|, \\ NAN, & \text{if } i(a) \text{ is undefined for } st(\mathcal{S}) \\ 100 * \phi/|st(\mathcal{S})|, & \text{else } (\phi = |st(\mathcal{S})| - |st(\mathcal{S}')|). \end{cases}$$

The above formulation introduces a heuristic that helps to break ties when we have a set of merging inputs. The heuristic step considers the number of merged states and promotes the input that is causing more states to merge. Note that when $|S'|$ is equal to $|S|$, the reward is -1, this is a step to prevent the agent from introducing redundant inputs to SS. Moreover, with immediate reward NAN, the proposed algorithm can derive SSs from partial systems without worrying about constructing SSs with undefined inputs. However, the algorithm may generate longer SSs when the underlying system is partial. This is due to the fact that the length of a shortest SS for a complete system is bounded by $O(n^2)$ [26] and for a partial system the bound is $O(n^2 * 4^{n/3})$ [65].

Finally, note that the for a given SS construction problem instance, the constructed MDP is finite. We formally state this property in the Corollary below

Corollary 1. *Let A be an automaton, the MDP P constructed from A for constructing an SS is a finite MDP.*

B. The Algorithm

Before going into the details of the algorithm we will introduce some basic concepts that the algorithm uses. The proposed algorithm uses an ϵ-greedy approach [56]. Using this approach we addressed the exploration and the exploitation trade off. With probability ϵ the agent chooses inputs randomly. Otherwise it selects inputs according to the $\mathcal{Q}$-values, which are computed using the standard $\mathcal{Q}$-value function given in Formula 1 where the next state ($\mathcal{S}'$) is computed using the δ function, i.e., $\mathcal{S}' = \delta(st(\mathcal{S}), i(a))$. The learning rate and future reward discount values were both set to 0.9, i.e., $\alpha = 0.9$ and $\gamma = 0.9$[2].

The algorithm receives an automaton A and an upper-bound on the number of episodes[3] (E) as its inputs. Then it constructs the $\mathcal{Q}$-graph. The construction of the Q graph is done by introducing a single $\mathcal{Q}$-node: a node that is associated with the set of states $n_{\mathcal{S}} = S$ of A with random $\mathcal{Q}$-values. (Lines 1-3 of Algorithm 1). The graph then gradually grows by introducing new nodes while the agent explores the environment. Once the initial node has been created, the algorithm initialises the graph with an empty input sequence and a pointer to the initial node ($initialNode$) that will be used when the algorithm wants to reach the first node of the $\mathcal{Q}$-graph i.e., when it picks an undefined input. The algorithm also sets the *episode counter* to 0 (Line 4 of Algorithm 1).

Afterwards it enters a loop that ends when (i) it finds an SS, i.e., it reaches a node n' such that $|n'_{\mathcal{S}}| = 1$, or (ii) the maximum number of episodes has been reached. Note that in a finite MDP the $\mathcal{Q}$-learning algorithm converges after a number of episodes, i.e., runs [56]. However, not all automata possess an SS. When there is no SS, the agent will repeatedly compute ω and never succeed. As a result of this observation, the algorithm requires an upper-bound on the number of episodes as input (E).

[2]The α and β values were manually set based on controlled experiments during which we measure the total rewards gained. This is then followed by the experiments. That is hyperparameters α and β were constant throughout the experiments.

[3]Throughout the experiments E was set to 1,000,000 episodes.

```
Input: Automaton $A = (S, \Sigma, H)$ such that $|S| > 1$, $E$
Output: An SS $\omega$ for $A$
begin
1     Initiate a $\mathcal{Q}$−node $n$ such that $n_{\mathcal{S}} \leftarrow st(S)$.
2     foreach $x \in \Sigma$ do
3         Assign a random value to $\mathcal{Q}(n_{\mathcal{S}}, i(x))$ of $n$.
      end
4     $InitialNode \leftarrow n$, $n' \leftarrow n$, $N \leftarrow N \cup n$, $\omega \leftarrow \varepsilon$, $e \leftarrow 0$.
5     while $e < E$ do
6         foreach $x \in \Sigma$ do
7             if $n''$ with $n''_{\mathcal{S}} = \delta(st(n'_{\mathcal{S}}), x)$ does not exist in $N$
               then
8                 Introduce $n''$ to $N$ such that
                    $n''_{\mathcal{S}} = \delta(st(n'_{\mathcal{S}}), x)$ having random
                    $\mathcal{Q}$-values.
9                 Introduce an edge from $n'$ to $n''$ labelled with
                    $i(x)$.
              end
          end
10        $e \leftarrow e + 1$, Pick a random value $r$ in the range $(0, 1]$.
11        if $r \leq \epsilon$ then
12            Select a random input $x$, update $n'$, and $\mathcal{Q}$-values.
          end
13        else
14            Select $x$ using $Q(n_{\mathcal{S}'}, i(x))$, update $n'$, and
                $\mathcal{Q}$-values.
          end
15        if $R(n'_{\mathcal{S}}|i(x)) = NAN$ then
16            $n' \leftarrow InitialNode$, $\omega = \varepsilon$.
          end
17        else
18            $\omega \leftarrow \omega \cdot x$.
19            if $|n'_{\mathcal{S}}| = 1$ then
20                return $\omega$.
              end
          end
      end
21    return $\varepsilon$.
end
```

Algorithm 1: The $\mathcal{Q}$-Synch algorithm.

At each iteration, the algorithm picks the current node (n'), extracts the set of states $n_{\mathcal{S}'}$ and it checks if all the adjacent nodes of n' are in N i.e., for each input $x \in \Sigma$ it checks if an adjacent node n'' is included in the $\mathcal{Q}$-Graph. If not, then it introduces the missing $\mathcal{Q}$-nodes with random $\mathcal{Q}$-values, and introduces the edge information (Lines 6-9 of Algorithm 1).

After this the algorithm increments e, updates the input symbol (x), the current $\mathcal{Q}$-node (n'), and $\mathcal{Q}$ values according to the ϵ-greedy method (Lines 10-14 of Algorithm 1).

If the immediate reward is NAN, then the algorithm clears ω, sets the current node as the initial node, and repeats the process. Otherwise, it appends the input to ω and checks whether the current node is associated with a singleton set or not. If so, the algorithm returns ω as the SS (Lines 15-20 of Algorithm 1). Otherwise it repeats the above mentioned process. If the upper-bound on the number of episodes is reached and no SS has been found then the algorithm terminates and returns an empty ω.

The above algorithm can generate an SS from a given A if and only if A has an SS and a suitably large episode value E is provided.

Proposition 1. *The Q-Synch algorithm can construct an SS from an automaton A if and only if A has one and E is sufficiently large.*

Proof. $\rightarrow$ This part follows from the Corollary 1, and the fact that Q-learning algorithm finds the optimal policy in finite MDPs [66].

$\leftarrow$ Now assume that the Q-Synch algorithm returns a non-empty input sequence ω but this is not a synchronising sequence. We consider two cases (i) there exists $s, s' \in S$ such that $\delta(s, \omega) \neq \delta(s', \omega)$ and (ii) there exist $s \in S$ such that $\delta(s, \omega)$ is not defined.

Since ω is non-empty the algorithm must return when a singleton set is reached. Therefore $|\delta(S, \omega)| = 1$ and (i) cannot be true. Next, assume that ω is in the form of $\omega' x \omega''$ where ω' and ω'' are sequences. Let us assume that $\delta(s, \omega') = s'$ but $\delta(s', x)$ is not defined. We need to consider two sub-cases: (a) ω' is an SS for A, and (b) ω' is not an SS for A. Note that the algorithm returns as soon as it reaches an SS, so (a) cannot be true. If (b) happens then the algorithm should clear ω and cannot return a non-empty sequence. Therefore (ii) cannot be true. Hence the result follows. $\square$

IV. Experiments

In this section, we provide the details of the controlled experiments conducted to answer our research questions. We first recall the research questions and the evaluation criteria, focusing on which aspects of the algorithms were compared. This is then followed by a description of the experimental subjects. Next, we outline the benchmark algorithms used as a baseline and the experiment environment used. Finally, we provide the results of the experiments.

A. Research Questions and Evaluation Criteria

The following research questions were posed at the outset (in Section 1):

- RQ 1 Is our proposed Q-learning algorithm more efficient and scalable than the state of the art algorithms?
  - RQ 1.1 Is the growth of the execution time more modest in the Q-learning algorithm?
  - RQ 1.2 How scalable can a Q-learning algorithm be? By fixing the amount of memory and execution time, can a Q-learning algorithm generate SSs from specifications that cannot be processed by the state of the art SS generation algorithms?
  - RQ 1.3 How does the memory usage behaviour of the Q-learning algorithm compare to state of the art SS generation algorithms?
- RQ 2 Is our proposed Q-learning algorithm more effective in generating shorter synchronising sequences for a larger class of system models?
  - RQ 2.1 Does the Q-learning algorithm generate shorter SSs (relative to execution time and memory consumption) than the state of the art SS generation algorithms?
  - RQ 2.2 Can the Q-learning algorithm be extended to mitigate the above-mentioned generalisation problem?

To answer these questions, we consider a number of evaluation criteria. The first one is the *time* an algorithm requires to generate an SS, with a faster algorithm being better. The second criterion is related to the *length* of the SSs generated by the algorithms. Since there is a cost associated with the application of an SS, we say that an algorithm is better than others if it generates a shorter sequence for a given automaton. The last criterion relates to the scalability of the algorithms. By scalability, we refer to two different aspects of the algorithms. First, we consider the maximum *number of model states* that the algorithm can process in a given time. Second, we considered the *memory* used by the algorithm while constructing SSs. So a scalable algorithm is the one that can process larger automata and requires less memory than others.

B. Experiment subjects

We used two sets of automata in the experiments. The sets ($S1$ and $S2$) contained randomly constructed synthetic automata, where $S1$ had completely specified automata and $S2$ contained partially specified automata. Synthetic automata were constructed using the following procedure.

Let n be the number of states and p be the number of inputs. To generate a completely specified automaton with n states and p inputs, we first generated a graph with n nodes (states) and for each node (state) we randomly generate p adjacent nodes. Then we checked whether the resultant automaton has an SS. If so, we kept it, we discarded it otherwise.

If the underlying automaton is to be partial then we again generated a graph with n nodes but this time for each node we picked k adjacent nodes from the graph where k was picked randomly in the range $[1, p]$. If the underlying automaton had an SS then we stored the automaton. We used each (n, p) pair in which $n \in \{32, 64, 128, \ldots, 131072\}$ and $p \in \{10, 16, 22\}$. For each such (n, p) pair, we randomly generated 100 automata for $S1$ and another 100 automata for $S2$, resulting in a total of 7800 automata.

In order to complement the experiments, we also used a specification of a real software *Engine Status Manager* (ESM). An ESM is a piece of control software that is used to manage the status of the engine in Océ printers and copiers (a subsidiary of Canon). This example is chosen, because its structure and behaviour is representative of embedded control software [67]. Moreover, the ESM model was not retrieved from its designers/developers but it was learned by another piece of software LearnLib [68] and rigorous verification has confirmed that the behavioural model is indicative of the actual system. The ESM model is partial and has 77 inputs and 3410 states.

C. Benchmark algorithms and experiment environment

For $S1$, we compared the Q-synch algorithm with the fastest sequential algorithm, algorithm called the *The Greedy Method* [26]. There are other more recent algorithms such as FastSynchro and SynchroP that can find shorter SSs than the Greedy approach; however the computational complexities of these algorithms are much worse than the Greedy algorithm and are $O(n^4|\Sigma|)$ and $O(n^5|\Sigma|)$ respectively [30]. An algorithm based on a SAT solver is provided in [69] however this algorithm is slow and cannot process large automata. For $S2$, we compared the $\mathcal{Q}$-synch algorithm against the only existing algorithm reported in the literature, the parallel brute-force (parallel BF) SS generation algorithm [15].

The sequential algorithms were implemented in C++, and compiled using Microsoft Visual Studio, edition 2012. The parallel BF algorithm was implemented in CUDA 6.0 using compute capability 2.1. The source-code, the constructed SSs, the automata in sets $S1$, $S2$, and the ESM are publicly available[4]. The computer used in the experiments had an Intel $I7-3630$QM CPU at 2.0GHz with 8GB RAM equipped with an NVIDIA Geforce 610M with 2GB memory. The operating system used was 64-bit Windows 7[5].

D. Results

In order to evaluate the relative performance of the algorithms, for each automaton A, we separately computed SSs using the proposed algorithm and the benchmark algorithms. We recorded the generated SS, the execution time, and memory required by the algorithms. Throughout the experiments, we set the execution time limit to one hour and set the memory limit to 1GB of RAM.

We ran the Greedy Algorithm with automata from set $S1$. The Greedy Algorithm could generate SS when $n \leq 512$. When $n > 512$ the memory requirement exceeded the given limit. We ran the parallel BF algorithm with automata from set $S2$, and the parallel BF algorithm could not compute SSs when $n > 64$ within the given time limit.

In the rest of this section we discuss the results in greater detail. We used R for statistical tests and to generate graphs [70][6].

1) Time comparison: We provided the time comparison results conducted on $S1$ in Figure 2a. The y-axis values give the mean ratio of the time taken by the $\mathcal{Q}$-Synch algorithm to the time taken by the Greedy Method. The results are promising and show that the proposed $\mathcal{Q}$-synch algorithm is 500 times faster than the Greedy algorithm on average. We also observe that as the number of states increases the difference increases. When $n = 512$ the $\mathcal{Q}$-Synch algorithm is about 1000 times faster than the Greedy method.

To investigate the results further, we conducted a statistical effect size analysis through computing Cohen's distance d [71], using the R tool [70]. The results regarding to the effect sizes for execution time are given in Table I. The statistical analyses indicate that the effect size between the population's is large.

We provide the time comparison results conducted on $S2$ in Figure 3a. Again we observe that the results are promising. The proposed $\mathcal{Q}$-synch algorithm was 1000 times faster than the parallel BF algorithm on average and when $n = 64$ the proposed algorithm was 10000 times faster on average. The result of Cohens' d analysis is given in Table IV. Results again indicate that the effect size between the populations is large.

2) Length of constructed SSs: The length comparison results for the SSs generated for $S1$ are given in Figure 2b. Similar to before, the y axis denotes the ratio of the lengths (length of an SS constructed by the Greedy method/ length of an SS constructed by the $\mathcal{Q}$-Synch algorithm); therefore, the higher the value, the better the $\mathcal{Q}$-Synch algorithm. The results indicate that the $\mathcal{Q}$-Synch algorithm constructs shorter SSs (30% shorter on average) than the Greedy method. The difference appears to plateau at around 30% and does not change with the number of states and inputs. The result of effect size analysis is given in Table II. The results also suggest that the populations effect size is slightly different. Regardless of the number of inputs and the number of states, the median for the length of SSs generated by the Greedy method is slightly higher than the median of the length of the SSs generated by the $\mathcal{Q}$-Synch algorithm.

The results conducted on $S2$ are given in Figure 3b. This time the y axis denotes the averages of the ratio of the lengths of SSs constructed by the parallel BF method to the lengths of SSs constructed by the $\mathcal{Q}$-Synch algorithm. Since the parallel BF algorithm is a brute-force algorithm it finds one of the shortest SS for each automaton from set $S2$. The SSs generated by the $\mathcal{Q}$-Synch algorithm are 38% longer than the SSs generated by the parallel BF algorithm on average. One promising observation is that the difference seems to plateau at around 38% and does not grow with the number of states or inputs. The Cohen's d analysis also indicates that the median of the lengths of the SSs constructed by the parallel BF algorithm is smaller the median of the lengths of the SSs constructed by the $\mathcal{Q}$-Synch algorithm (Table V).

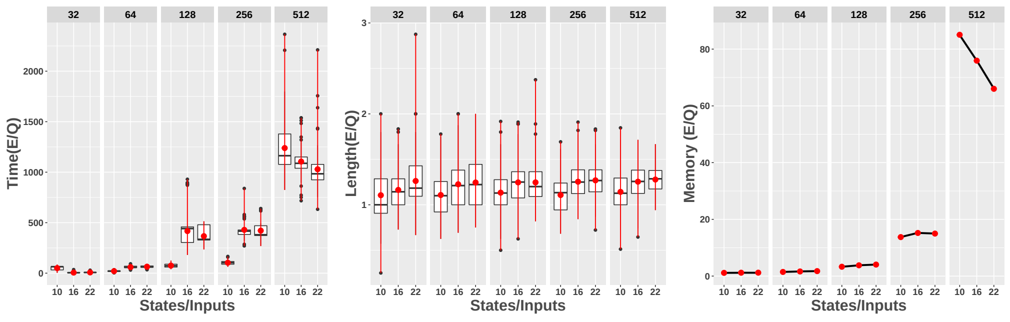
3) Memory requirements: The memory requirements for automata in $S1$ are given in Figure 2c. The y axis provides the ratio of the memory required by the Greedy algorithm to the memory required by the $\mathcal{Q}$-Synch algorithm. We used `WorkingSetSize` property of `PROCESS_MEMORY_COUNTERS` of Windows API to get the memory information. The figure indicates that the Greedy algorithm requested more memory than the $\mathcal{Q}$-Synch algorithm (30 times more memory on average) and the difference increases with the number of states. When $n = 512$ the Greedy algorithm requires 76 times more memory than the proposed algorithm on average and when $n = 1024$ the Greedy algorithm failed to generate SSs within the given memory limit. Moreover, considering the effect size analysis in Table III, we see the results of effect size analysis and that the results are conclusive, the effect size between the two populations is large.

The memory consumption comparison for the parallel BF method and $\mathcal{Q}$-Synch algorithm is given in Figure 3c. Again, the y axis gives the ratio of the amount of memory required by the parallel BF algorithm to the memory required by the

[4]Code and data are anonymously available at https://figshare.com/articles/software/CodeAndData/14478132

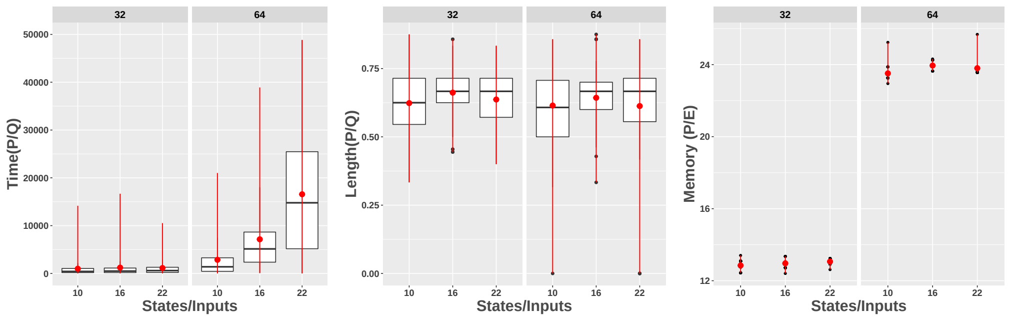
[5]Please note that in [15] the authors used a Tesla K40 GPU to conduct the experiments which is about 10,000 times faster than the GPU card used in these experiments.

[6]The R source codes and the data are anonymously published in https://figshare.com/s/cbf85c54a1ff11674374



(a) (b) (c)

Fig. 2: Comparison of the performances of the Greedy and the $\mathcal{Q}$-Synch algorithms on dataset $S1$. Figure 2a summarises the ratio of the time required to construct SSs, Figure 2b summaries the ratio of the lengths of the SSs, and Figure 2c summarises the ratio of memory required to construct SSs.



(a) (b) (c)

Fig. 3: Comparison of the performances of the parallel BF and the $\mathcal{Q}$-Synch algorithms on dataset $S2$. Figure 3a summarises the ratio of the time required to construct SSs, Figure 3b summaries the ratio of the lengths of the SSs, and Figure 3c summarises the ratio of memory required to construct SSs.

$\mathcal{Q}$-Synch algorithm. The results are similar, with the proposed algorithm requiring less memory (18% on average) than the parallel BF algorithm. Similar to before, we can observed that the ratio increases as the number of states increases. We again complement the analysis using the Cohen's d metric (Table VI). Results suggest that the effect size is large, meaning that the populations are different.

n	p	Cohens'd
32	10	13.863
	16	1.837
	22	2.488
64	10	8.123
	16	19.045
	22	33.176
128	10	30.253
	16	47.210
	22	63.369
256	10	60.639
	16	74.934
	22	83.587
512	10	30.491
	16	42.241
	22	46.386

TABLE I: Cohens'd analysis on the amount of time required by the algorithms on $S1$.

n	p	Cohens'd
32	10	0.200
	16	0.802
	22	0.998
64	10	0.301
	16	0.945
	22	1.204
128	10	0.588
	16	1.173
	22	1.518
256	10	0.475
	16	1.466
	22	1.648
512	10	0.618
	16	1.427
	22	2.272

TABLE II: Cohens'd analysis on the lengths of SSs constructed by the algorithms on $S1$.

n	p	Cohens'd
32	10	43.385
	16	32.861
	22	6.456
64	10	1394.707
	16	33.283
	22	1215.009
128	10	1599.230
	16	452.060
	22	481.147
256	10	1005.881
	16	852.319
	22	825.083
512	10	2307.331
	16	2426.511
	22	1901.071

TABLE III: Cohens'd analysis on the memory requirements of the algorithms on $S1$.

4) Results on the benchmark automaton: Recall that the ESM specification is partially specified. To conduct the experiments we generated two versions of the ESM specification (v_1,v_2). v_1 was generated by completing the missing transitions to obtain a completely specified model. Completing a partial model is a well known approach in MBT [64], [72]. To complete the missing transitions, we first introduced an `error state` and for each state s and for each missing transition labeled by an input x, we introduced a transition from s to the `error state` with input x. The second version of ESM was the original partial version. We then investigated the model and discovered that the ESM model does not possess an SS. This is because state pairs $(s524, s721)$, $(s70, s71)$, $(s304, s3088)$, $(s344, s2933)$, and $(s1080, s3258)$ are *absorbing pairs*. An absorbing pair is a pair of states (s_i, s_j) such that for each input $x \in \Sigma$ we have that $\delta(s_i, x) = s_j$ and $\delta(s_j, x) = s_i$. That is, no transitions leave these pairs. In order to be able to use the automaton in the experiments, for each of these 10 states, we modified transitions labelled with a common input

n	p	Cohens'd
32	10	0.870
	16	0.751
	22	1.221
64	10	0.887
	16	1.125
	22	1.511

TABLE IV: Cohens'd analysis on the amount of time required by the algorithms on $S2$.

n	p	Cohens'd
32	10	-2.304
	16	-2.586
	22	-3.058
64	10	-2.393
	16	-2.748
	22	-2.866

TABLE V: Cohens'd analysis on the lengths of SSs constructed by the algorithms on $S2$.

n	p	Cohens'd
32	10	89.739
	16	105.909
	22	115.326
64	10	132.892
	16	235.355
	22	260.244

TABLE VI: Cohens'd analysis on the memory requirements of the algorithms on $S2$.

symbol $I21.1$ such that they all merge at state $s524$.

After the above modifications were made, we provided v_1 to the Greedy and the $\mathcal{Q}$-Synch algorithms and provided v_2 to the parallel BF and the $\mathcal{Q}$-Synch algorithms as input. In the experiments, the Greedy and the parallel BF algorithms could not generate SSs. As in the case of randomly generated automata, the Greedy algorithm failed to construct an SS due to memory problem and similarly the parallel BF algorithm could not finish computing an SS within one hour. However, the $\mathcal{Q}$-Synch algorithm did generate an SS for these specifications. The lengths of the SSs for v_1 and v_2 were 356 and 374 respectively and they reset the system to state $s524$ as expected. The $\mathcal{Q}$-Synch algorithm was able to generate the sequences in 351 milliseconds and used 326 MBs of RAM on average.

5) Scalability: As indicated earlier, we investigated the scalability of the algorithms with respect to (i) the maximum number of states, and (ii) amount of memory required to construct SSs. In Figure 4, we provided the average times required to construct SSs from $S1$ and $S2$ using the $\mathcal{Q}$-Synch algorithm. The proposed algorithm is able to construct SSs from specifications with 131072 states and 22 inputs. Since the Greedy and the parallel BF algorithms could only generate SSs when $n \leq 512$ and $n \leq 64$, respectively, with respect to number of states the proposed algorithm is 256 times more scalable than the Greedy algorithm. Moreover, the $\mathcal{Q}$-Synch algorithm is 2048 times more scalable than the parallel BF algorithm.

In Figure 5, we present the results for $S1$. Here, each value is the mean memory required for the Greedy algorithm (Figure 5a) to process the given size of automata in $S1$. Similarly, Figure 5b gives the mean memory usage by the implementation of the parallel BF algorithm for $S2$. Moreover, in Figure 6, we give the mean memory required, in MB, by the proposed algorithm to construct SSs ($S1$ in Figure 6a, and $S2$ in Figure 6b).

The memory requirement of the proposed algorithm does not grow as fast as that of the other algorithms. To investigate this, as the $\mathcal{Q}$-learning algorithm uses $\mathcal{Q}$-nodes, we checked the mean number of $\mathcal{Q}$-nodes created while computing SSs. In Figure 7, we provided these values (mean number of $\mathcal{Q}$-nodes constructed by the proposed algorithm): Figure 7a for $S1$, and Figure 7b for $S2$. We observe that the $\mathcal{Q}$-Synch algorithm is very economic in the sense that it generates $\mathcal{Q}$-nodes tentatively. This is important and implies that the $\mathcal{Q}$-learning algorithm selects its inputs wisely. If this was not the case, in each episode the agent would select different inputs and introduce new nodes to the $\mathcal{Q}$-graph.

V. Discussions

In this section, in light of the conducted experiments, we discuss the answers to the research questions. This is then followed by an analysis of some threats to validity.

A. Answers to the research questions

RQ 1 Is our proposed $\mathcal{Q}$-learning algorithm more efficient and scalable than the state of the art algorithms?

Answer 1: The results of the experiments suggest that our algorithm is more efficient and scalable than the most scalable algorithms reported in the literature. However, we also noted that the number of inputs and states have negative impact on the scalability of the proposed method.

RQ 1.1 Is the growth of the execution time more modest in the $\mathcal{Q}$-learning algorithm?

Answer 1.1: The results indicate that the time requirement of the proposed algorithm is modest and its growth rate is much slower than the state of the art SS generation algorithms. Results indicate that the proposed algorithm is 500/1000 times faster than the fastest sequential/parallel algorithm on average.

RQ 1.2 How scalable can a $\mathcal{Q}$-learning algorithm be? By fixing the amount of memory and execution time, can a $\mathcal{Q}$-learning algorithm generate SSs from specifications that cannot be processed by the state of the art SS generation algorithms?

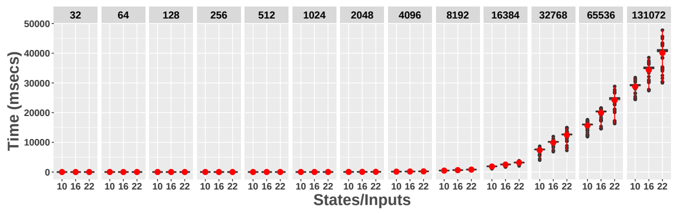
Answer 1.2: The proposed algorithm can quickly generate SSs while using less memory. The experimental results show that the proposed method can process 256 times larger specifications than the state of the art sequential algorithm. What is more, the proposed algorithm is 2048 times more scalable than the state of the art GPU based massively parallel algorithm. Finally, the experimental results suggest that when there are limited computation resources, the proposed algorithm can generate SSs from real specifications where other algorithms cannot.

RQ 1.3 How does the memory usage behaviour of the $\mathcal{Q}$-learning algorithm compare to state of the art SS generation algorithms?

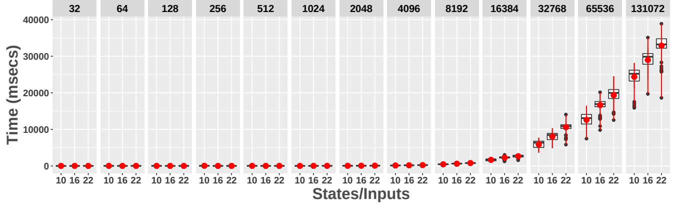
Answer 1.3: The proposed algorithm requires neither a product automaton nor a preset $\mathcal{Q}$-table to be built; therefore, the memory requirement of the proposed method grows much slower than the state of the art sequential SS generation method. Experimental study indicates that the Greedy algorithm requires 30 times more memory than the proposed algorithm on average. The parallel BF SS generation algorithm on the other hand requires 18 times more memory than the proposed algorithm.

RQ 2 Is our proposed $\mathcal{Q}$-learning algorithm more effective in generating shorter synchronising sequences for a larger class of system models?

Answer 2 The result of the experiments indicate that the proposed algorithm is more effective in generating short SSs. We compared the results with the Greedy algorithm and the results suggested that the proposed algorithm finds SSs that are 30% shorter on average.

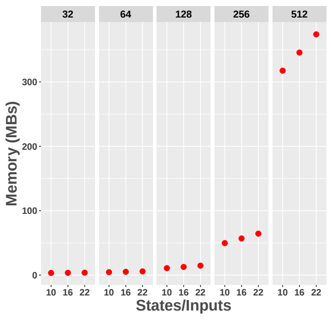


(a)

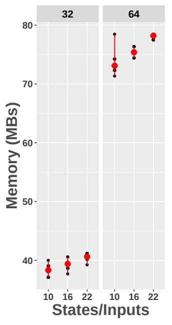


(b)

Fig. 4: The average amount of time spent to generate SSs from $S1$ (Figure 4a) and from $S2$ by using the $\mathcal{Q}$-Synch algorithm (Figure 4b).



(a)



(b)

Fig. 5: The averages of memory requirement to generate SSs from $S1$ using The Greedy Method (Figure 5a) and from $S2$ using the parallel BF algorithm (Figure 5b).

RQ 2.1 Does the $\mathcal{Q}$-learning algorithm generate shorter SSs (relative to execution time and memory consumption) than the state of the art SS generation algorithms?

Answer 2.1: The proposed algorithm can generate shorter SSs and consumes much less memory and time than the fastest sequential SS generation algorithm does. Moreover the proposed algorithm can compute SSs that are comparable in length with the parallel BF SS generation algorithm.

RQ 2.2 Can the $\mathcal{Q}$-learning algorithm be extended to mitigate the above-mentioned generalisation problem?

Answer 2.2: We introduced a new $\mathcal{Q}$-learning framework in which we abandoned the idea of keeping the entire search space in a preset form. Instead, we employ a method where the search space grows as the agent interacts with the environment. This allows us to represent the search space using a $\mathcal{Q}$-graph. We used this formalism in the classical $\mathcal{Q}$-learning algorithm. To our knowledge, this is new and experimental studies showed that it allows agents to learn $\mathcal{Q}$-values from large environments.

Experimental evaluation indicates that using the $\mathcal{Q}$-graph formalism we can solve problems using a search space whose size is a fraction of the size of the search space needed in the classical $\mathcal{Q}$-table setting. For example, as presented in Figure 7a, and Figure 7b, our method needs 70 $\mathcal{Q}$-nodes to derive an SS from an automaton that has $|S| = 131072$ states and $|\Sigma| = 22$ inputs on average. If we were using a tabular setting, we would generate a table having $pow(S) * |\Sigma| = (2^{131072} - 1) * 22$ cells which would not be possible.

B. Threats to validity

There are number of threats to the validity of the experiments. The first threat is to *generalisability* which originates from the fact that the experimental subjects may not be representative of real systems. The use of randomly generated automata clearly introduces such a threat and we addressed this by creating two versions of the specification of *Engine Status Manager* software, which has 3410 states and 77 inputs. Importantly, the experimental results obtained with these real-world models are similar to those obtained with randomly generated automata.

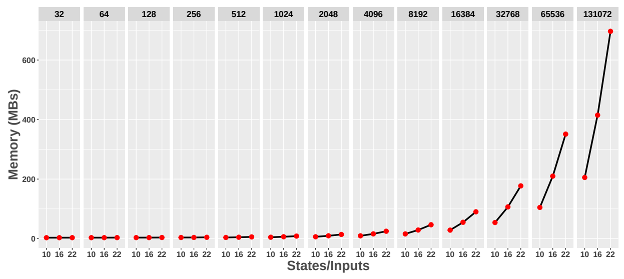
There are also threats to internal validity and the possibility that one or more of the implementations were incorrect. To reduce this threat, we applied unit testing in the development cycle. Moreover, when a sequence (ω) was generated by an algorithm, we checked that the generated sequence was an SS. To achieve this, when an SS ω was computed for an automaton A, we randomly selected an initial state (s) of A and, starting from s, we applied ω to find the reset state s'. Clearly s' should be the state that the A reaches regardless of the initial state from which ω is applied. To confirm this, for every state s'' of A, we checked that the application of ω in s'' took A to s'. Throughout the experiments we did not encounter a case where the underlying automaton failed to reach the reset state.

Finally, there is potential to misinterpret the results obtained from the experiments. To address this threat, we validated our results by conducting Cohen's d effect size analysis.

VI. Conclusion

Model based testing (MBT) is an increasingly important type of software testing. Most MBT techniques, require some method that brings the system under test (SUT) to a specific initial state in order for a test sequence to be applied to the SUT. This requirement can be fulfilled by a *synchronising sequence (SS)* [38]. The length of an SS used affects the cost of test execution and so there has been long standing interest in the problem of finding a short SS [21]. However, the problem of generating short SSs is known to be **NP**-*hard*.

The other motivation for the work described in this paper comes from the fact that previous work has developed a variety


(a)

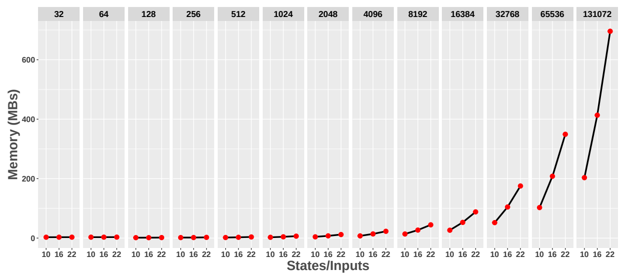
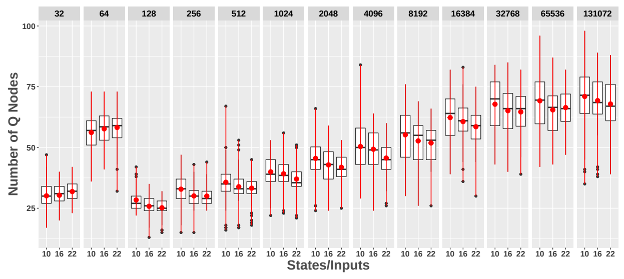

(b)

Fig. 6: The averages of the memory requirement of the Q-Synch algorithm when generating SSs from $S1$ (Figure 6a) and from $S2$ (Figure 6b).


(a)

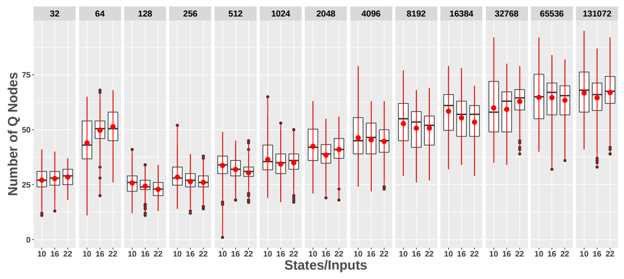

(b)

Fig. 7: The averages of the number of Q-nodes generated while constructing SSs from $S1$ (a) and $S2$ (b) by using the Q-Synch algorithm.

of successful automated test generation methods based on Reinforcement Learning (RL) [7], [47], [52]. In this paper, we proposed a new Q-learning algorithm to derive synchronising sequences. The proposed method introduces the notion of Q-graph which, instead of holding the entire search space in memory, allows the search space to be expanded on-the-fly. Experimental results indicate that the proposed method is more efficient and effective in generating SSs than the state of the art SS generation methods.

There are a number of lines of future work. First, we will investigate the implications of the introduced Q-learning framework on other state-exploration problems in model-based testing and beyond. Instead of holding the entire search space in a table, our Q-learning framework allows the search space to be represented as a graph and therefore uses less memory space, allowing learning from very large search spaces. This study might lead to new RL algorithms. Besides, there may be scope to investigate other RL approaches for deriving SSs. Further potential directions include extensions of the framework to probabilistic automata with unknown transition probabilities. Moreover, it would be interesting to study the effect of shorter SS on testing. Since the impact will depend on the test technique used, a systematic evaluation of different test methods should be carried out.

Finally, the experimental results suggest that as the number of states and inputs of the automata grow, the time and memory requirements of the method increase. Although this is unsurprising, we plan to explore approaches, such as parallelisation, that might allow the technique to scale further.

Acknowledgements We have been partially supported by the UKRI Trustworthy Autonomous Systems Node in Verifiability, Grant Award Reference EP/V026801/1.

References

[1] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice," in *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, 1999, pp. 285–294.

[2] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Trans. Software Eng.*, vol. 4, no. 3, pp. 178–187, 1978.

[3] 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.

[4] 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.

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

[6] 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.

[7] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 301–310.

[8] D. Damasceno, M. R. Mousavi, and A. Simão, "Learning to reuse: Adaptive model learning for evolving systems," in *Integrated Formal Methods - 15th International Conference, IFM 2019*, ser. Lecture Notes in Computer Science, vol. 11918. Springer, 2019, pp. 138–156.

[9] 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.

[10] 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, 1964, pp. 95–110.

[11] G. V. Jourdan, H. Ural, and H. Yenigun, "Reduced checking sequences using unreliable reset," *Inf. Process. Lett.*, vol. 115, no. 5, pp. 532–535, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.ipl.2015.01.002

[12] R. Boute, "Distinguishing sets for optimal state identification in checking experiments," *IEEE Transactions on Computers*, vol. 23, no. 8, pp. 874–877, 1974.

[13] M. Vasilevskii, "Failure diagnosis of automata," *Cybernetics*, vol. 9, no. 4, pp. 653–665, 1973. [Online]. Available: http://dx.doi.org/10.1007/BF01068590

[14] B. K. Natarajan, "An algorithmic approach to the automated design of parts orienters," in *FOCS*, 1986, pp. 132–142.

[15] U. C. Türker, "Parallel brute-force algorithm for deriving reset sequences from deterministic incomplete finite automata," *Turkish Journal of Electrical Engineering & Computer Sciences*, vol. 27, pp. 3544–3556, 09 2019.

[16] U. C. Türker and H. Yenigün, "Complexities of some problems related to synchronizing, non-synchronizing and monotonic automata," *International Journal of Foundations of Computer Science*, vol. 26, no. 01, pp. 99–121, 2015.

[17] D. S. Ananichev and M. V. Volkov, "Synchronizing monotonic automata," *Theor. Comput. Sci.*, vol. 327, no. 3, pp. 225–239, 2004.

[18] ——, "Synchronizing generalized monotonic automata," *Theor. Comput. Sci.*, vol. 330, no. 1, pp. 3–13, 2005. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2004.09.006

[19] A. N. Trakhtman, "Some results of implemented algorithms of synchronization," in *10th Journees Montoises d'Inform*, 2004.

[20] ——, "Synchronization of some DFA," in *Theory and Applications of Models of Computation, 4th International Conference, TAMC 2007, Shanghai, China, May 22-25, 2007, Proceedings*, 2007, pp. 234–243.

[21] ——, "Modifying the upper bound on the length of minimal synchronizing word," in *Fundamentals of Computation Theory - 18th International Symposium, FCT 2011, Oslo, Norway, August 22-25, 2011. Proceedings*, 2011, pp. 173–180.

[22] ——, "The length of a minimal synchronizing word and the Černy conjecture," *CoRR*, vol. abs/1405.2435, 2014.

[23] ——, "Some aspects of synchronization of DFA," *J. Comput. Sci. Technol.*, vol. 23, no. 5, pp. 719–727, 2008.

[24] M. V. Volkov, "Synchronizing automata and the černy conjecture," in *LATA*, 2008, pp. 11–27.

[25] ——, "Synchronizing automata preserving a chain of partial orders," *Theor. Comput. Sci.*, vol. 410, no. 37, pp. 3513–3519, 2009.

[26] D. Eppstein, "Reset sequences for monotonic automata," *SIAM J. Comput.*, vol. 19, no. 3, pp. 500–510, 1990.

[27] Y. Benenson, T. Paz-Elizur, A. Rivka, E. Keinan, Z. Livneh, and E. Shapiro, "Programmable and autonomous computing machine made of biomolecules," *Nature*, vol. 414, no. 6862, pp. 430–434, 2001. [Online]. Available: http://dx.doi.org/10.1038/35106533

[28] A. Roman, "New algorithms for finding short reset sequences in synchronizing automata," in *International Enformatika Conference, IEC'05, August 26-28, 2005, Prague, Czech Republic, CDROM*, 2005, pp. 13–17.

[29] O. Rafiq and L. Cacciari, "Coordination algorithm for distributed testing," *The Journal of Supercomputing*, vol. 24, no. 2, pp. 203–211, 2003.

[30] R. Kudlacik, A. Roman, and H. Wagner, "Effective synchronizing algorithms," *Expert Systems with Applications*, vol. 39, no. 14, pp. 11 746–11 757, 2012.

[31] A. Roman and M. Szykula, "Forward and backward synchronizing algorithms," *Expert Syst. Appl.*, vol. 42, no. 24, pp. 9512–9527, 2015.

[32] R. Raz and S. Safra, "A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP," in *STOC*, 1997, pp. 475–484.

[33] A. Roman, "Genetic algorithm for synchronization," in *Language and Automata Theory and Applications, Third International Conference, LATA 2009, Tarragona, Spain, April 2-8, 2009. Proceedings*, 2009, pp. 684–695.

[34] ——, "Synchronizing finite automata with short reset words," *Applied Mathematics and Computation*, vol. 209, no. 1, pp. 125–136, 2009.

[35] A. Kisielewicz, J. Kowalski, and M. Szykuła, "A fast algorithm finding the shortest reset words," *Computing and Combinatorics*, vol. 7936, pp. 182–196.

[36] S. Karahoda, O. T. Erenay, K. Kaya, U. C. Türker, and H. Yenigün, "Multicore and manycore parallelization of cheap synchronizing sequence heuristics," *J. Parallel Distributed Comput.*, vol. 140, pp. 13–24, 2020. [Online]. Available: https://doi.org/10.1016/j.jpdc.2020.02.009

[37] A. N. Trakhtman, "Modifying the upper bound on the length of minimal synchronizing word," *ArXiv e-prints*, 2011.

[38] A. Gill, *Introduction to The Theory of Finite State Machines*. McGraw-Hill, New York, 1962.

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

[40] C. Güniçen, E. Erdem, and H. Yenigün, "Generating shortest synchronizing sequences using answer set programming," *CoRR*, vol. abs/1312.6146, 2013. [Online]. Available: http://arxiv.org/abs/1312.6146

[41] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015. [Online]. Available: https://science.sciencemag.org/content/349/6245/255

[42] D. Chapman and L. P. Kaelbling, "Input generalization in delayed reinforcement learning: An algorithm and performance comparisons," in *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'91. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, p. 726–731.

[43] H. Almulla and G. Gay, "Learning how to search: Generating effective test cases through adaptive fitness function selection," *ArXiv*, vol. abs/2102.04822, 2021.

[44] S. Bauersfeld and T. Vos, "A reinforcement learning approach to automated GUI robustness testing," 2012.

[45] J. Eskonen, J. Kahles, and J. Reijonen, "Automating GUI testing with image-based deep reinforcement learning," *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pp. 160–167, 2020.

[46] L. R. Harries, R. S. Clarke, T. Chapman, S. V. P. L. N. Nallamalli, L. Özgür, S. Jain, A. Leung, S. Lim, A. Dietrich, J. M. Hernández-Lobato, T. Ellis, C. Zhang, and K. Ciosek, "Drift: Deep reinforcement learning for functional software testing," *ArXiv*, vol. abs/2007.08220, 2020.

[47] A. Reichstaller, B. Eberhardinger, A. Knapp, W. Reif, and M. Gehlen, "Risk-based interoperability testing using reinforcement learning," in *Testing Software and Systems*, F. Wotawa, M. Nica, and N. Kushik, Eds. Cham: Springer International Publishing, 2016, pp. 52–69.

[48] T. Vos, P. Tonella, I. Prasetya, P. M. Kruse, O. Shehory, A. Bagnato, and M. Harman, "The FITTEST tool suite for testing future internet applications," in *FITTEST@ICTSS*, 2013.

[49] X. Qin, N. Aréchiga, A. Best, and J. Deshmukh, "Automatic testing and falsification with dynamically constrained reinforcement learning," 2020.

[50] H. Yasin, S. Hamid, and R. Yusof, "Droidbotx: Test case generation tool for android applications using Q-learning," *Symmetry*, vol. 13, p. 310, 02 2021.

[51] M. Veanes, P. Roy, and C. Campbell, "Online testing with reinforcement learning," in *Formal Approaches to Software Testing and Runtime Verification*, K. Havelund, M. Núñez, G. Roşu, and B. Wolff, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 240–253.

[52] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang, "Predictive mutation testing," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 342–353. [Online]. Available: https://doi.org/10.1145/2931037.2931038

[53] Z. Pan and P. Mishra, "Automated test generation for hardware trojan detection using reinforcement learning," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 408–413. [Online]. Available: https://doi.org/10.1145/3394885.3431595

[54] Z. Pan, J. Sheldon, and P. Mishra, "Test generation using reinforcement learning for delay-based side-channel analysis," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–7.

[55] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992698

[56] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html

[57] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14236

[58] J. A. Boyan and A. W. Moore, "Generalization in reinforcement learning: Safely approximating the value function," in *Proceedings of the 7th International Conference on Neural Information Processing Systems*, ser. NIPS'94. Cambridge, MA, USA: MIT Press, 1994, p. 369–376.

[59] I. Podolak, A. Roman, M. Szykuła, and B. Zieliński, "A machine learning approach to synchronization of automata," *Expert Systems with Applications*, vol. 97, pp. 357–371, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417417308655

[60] A. Moore and C. Atkeson, "The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces," *Machine Learning*, vol. 21, 05 2000.

[61] H. Li, T. Chen, H. Teng, and Y. Jiang, "A graph-based reinforcement learning method with converged state exploration and exploitation," *Computer Modeling in Engineering & Sciences*, vol. 118, pp. 253–274, 02 2019.

[62] W. Smeenk, "Applying automata learning to complex industrial software," Master's thesis, Computer Science, 2012.

[63] S. C. Paiva, A. Simao, M. Varshosaz, and M. R. Mousavi, "Complete ioco test cases: A case study," in *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, ser. A-TEST 2016. New York, NY, USA: ACM, 2016, pp. 38–44. [Online]. Available: http://doi.acm.org/10.1145/2994291.2994297

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

[65] Z. Gazdag, S. Iván, and J. Nagy-György, "Improved upper bounds on synchronizing nondeterministic automata," *Inf. Process. Lett.*, vol. 109, no. 17, pp. 986–990, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.ipl.2009.05.007

[66] F. Melo and I. Ribeiro, "Convergence of Q-learning with linear function approximation," *2007 European Control Conference, ECC 2007*, 03 2015.

[67] W. Smeenk, J. Moerman, F. Vaandrager, and D. N. Jansen, "Applying automata learning to embedded control software," in *Formal Methods and Software Engineering*, M. Butler, S. Conchon, and F. Zaïdi, Eds. Cham: Springer International Publishing, 2015, pp. 67–83.

[68] B. Steffen, F. Howar, and M. Merten, *Introduction to Active Automata Learning from a Practical Perspective*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 256–296.

[69] H. Shabana and M. V. Volkov, "Using Sat solvers for synchronization issues in partial deterministic automata," in *Mathematical Optimization Theory and Operations Research*, I. Bykadorov, V. Strusevich, and T. Tchemisova, Eds. Cham: Springer International Publishing, 2019, pp. 103–118.

[70] P. Teetor, *R Cookbook*, 1st ed. O'Reilly, 2011.

[71] J. Cohen, *Statistical power analysis for the behavioral sciences*. Routledge, 1988.

[72] R. M. Hierons and U. C. Türker, "Distinguishing sequences for partially specified FSMs," in *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*, 2014, pp. 62–76.