Citation for published version (APA):

Citing this paper
Please note that where the full-text provided on King's Research Portal is the Author Accepted Manuscript or Post-Print version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version for pagination, volume/issue, and date of publication details. And where the final published version is provided on the Research Portal, if citing you are again advised to check the publisher's website for any subsequent corrections.

General rights
Copyright and moral rights for the publications made accessible in the Research Portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognize and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the Research Portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the Research Portal

Take down policy
If you believe that this document breaches copyright please contact librarypure@kcl.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 04. Feb. 2024
Accelerating Finite State Machine-Based Testing using Reinforcement Learning

Uraz Cengiz Türker, Member, IEEE, Robert M. Hierons, Khaled El-Fakih, Mohammad Reza Mousavi, and Ivan Y. Tyukin

Abstract—Testing is a crucial phase in the development of complex systems, and this has led to interest in automated test generation techniques based on state-based models. Many approaches use models that are types of finite state machine (FSM). Corresponding test generation algorithms typically require that certain test components, such as reset sequences (RSs) and preset distinguishing sequences (PDSs), have been produced for the FSM specification. Unfortunately, the generation of RSs and PDSs is computationally expensive, and this affects the scalability of such FSM-based test generation algorithms. This paper addresses this scalability problem by introducing a reinforcement learning framework: the Q-Graph framework for MBT. We show how this framework can be used in the generation of RSs and PDSs and consider both (potentially partial) timed and untimed models. The proposed approach was evaluated using three types of FSMs: randomly generated FSMs, FSMs from a benchmark, and an FSM of an Engine Status Manager for a printer. In experiments, the proposed approach was much faster and used much less memory than the state-of-the-art methods in computing PDSs and RSs.

Index Terms—Finite state machines, reset sequences, state identification sequences, reinforcement learning, Q-value function, software engineering/software/program verification, software engineering/test design, software engineering/testing and debugging.

1 INTRODUCTION

A utomated testing is an essential component for quality assurance and establishing trust in complex systems. Model-based testing (MBT) has been studied and used extensively as it enables automated test derivation. In MBT, the underlying system is usually modelled by state-based models. State-based models represent the input/output behaviour of the system and the change of internal state as a result of such behaviour. Finite state machines (FSMs), timed-finite state machines (tFSMs), extended finite state machines (EFSMs), and communicating extended finite state machines (C-EFSMs) are some of the most popular representation approaches deployed in MBT.

A model-based test generation technique derives tests from a state-based model, with the tests being applied to the system under test (SUT). Typically, tFSMs, EFSMs, and C-EFSMs are converted into equivalent FSM counterparts for test generation and numerous techniques have been proposed for deriving structured test cases from FSM models. In an FSM, an input \( x \) in a state \( s \) leads to an output \( y \) and next state \( s' \), with this defining a transition. FSM based techniques are appealing both because they can be automated and because, under certain assumptions, many of the techniques produce test cases with guaranteed fault detection ability [1], [2], [3], [4], [5].

Many model-based test generation techniques have been adopted in the industrial context [6]. A problem hampering the widespread use of such techniques is their efficiency for large-scale models. This challenge has become more prominent with the introduction of automata learning tools that can automatically derive state-based models from complex systems [7], [8], [9], [10]. In this paper, as part of the evaluation, we use such a model: the Engine Status Manager is an embedded program that manages the status of the engine in Oce printers and copiers (a subsidiary of Canon) [11]. This model has thousands of states, and systematically deriving tests with guaranteed fault detection capability from such large complex systems, is a challenge [12], [13], [14]. We address this scalability problem using reinforcement learning.

1.1 Problem definition and research questions

In FSM-based testing, the tester will typically bring the SUT to a specific initial state by executing a reliable reset function or applying a resetting sequence (RS) before the test [7], [8], [9], [10], [15]. In the absence of an RS, resetting the SUT may require manual configuration of the SUT or restarting the SUT and, therefore, is considered to be one of the most expensive steps in FSM-based testing [16]. This has led to research regarding algorithms that generate RSs from FSMs [12], [17], [18].

After the resetting phase, the tester uses a sequence of inputs and expected outputs to achieve a test purpose [15]. One crucial test purpose is to identify the existence of a one-to-one mapping between the states of the SUT and the specification FSM and to verify the correct implementation
of the transitions in the SUT [15]. Checking a transition requires further testing to verify that the state of the SUT reached by the transition matches the one in the specification FSM. State identification and transition verification use state identification sequences such as a Preset or Adaptive Distinguishing sequence (PDS, ADS), a Characterising Set [1], or a Unique Input Output sequence (UIO) [15], [19]. One advantage of using ADSs is that there is a polynomial time algorithm for finding an ADS, where it exists. However, the test environment may not allow adaptive testing; the tester cannot respond in real-time to outputs. In such scenarios, instead of an ADS, we may use PDSs. Therefore, many test generation algorithms require an RS and a PDS to construct a test [1], [16], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29]. However, the state-of-the-art approaches in deriving these sequences cannot process large FSMs. For example, the existing algorithm for deriving PDSs applies a brute-force search [30]. The state-of-the-art RS generation algorithms require general-purpose graphics processing units and cannot process FSMs having large input domains [12], [17], [18]. Therefore, to enable test case generation from large FSMs, we need scalable methods for deriving RSs and PDSs.

On the other hand, there are many classes of FSMs, representing different types of software systems. For example, if all states of a software system share a common input domain, then this system can be represented by a complete FSM; otherwise, the FSM is partial. In some software systems (partial or complete), state change occurs due to a timeout. Real-time software systems are examples where the state transition may happen due to a timeout. Such systems can be represented by a tFSMs [31], [32], [33] with a single clock. tFSMs have been used as models in many fields, such as in real-time scheduling and optimisation as well as model-checking, and there are tools for test-case generation from timed FSMs. An expressive version of them is supported by the Uppaal toolset [34]. Approaches for test derivation from tFSMs are investigated in many papers [31], [35], [36], [37]. A tFSM can be complete or partial and flattened into an FSM [33]. Thus, test derivation from a given tFSM, and the generation of related RSs and DSs, can be carried out using traditional FSM-based methods from the flattened FSM [36], [37]. However, flattening an FSM increases the state space size, leading to additional scalability issues, and to our knowledge, there are no RS and PDS generation methods for tFSMs.

Reinforcement Learning (RL) approaches are becoming a major tool in computational intelligence due to their exceptional problem-solving capabilities [38], [39]. In contrast to approaches that estimate the expected cumulative reward of being in a particular state, in Q-learning, we estimate the expected cumulative reward of taking a particular action in a particular state. Because of this nature, Q-learning is an appealing framework for test generation as it provides pairing between inputs and states. Based on this analogy, recently, we introduced a novel Q-learning algorithm [40] that has been shown to be effective in deriving synchronising sequences (SS) from complete (or partial) finite automata.

The work presented in this paper is motivated by the desire to create a Q-learning framework in which different types of test inputs can be derived from various types of FSMs. This makes it possible to represent reset sequence (RS) and state identification sequence (PDS) generation problems as objective functions that can be solved by the framework for complete (partial) timed (untimed) FSMs. We consider the following research questions:

**RQ-1:** Is it possible to define a general Q-learning framework that can compute resetting and state-verification sequences for different kinds of FSMs?

(a) What sort of reward functions can be used to derive resetting and state-verification sequences from different types of FSMs?

(b) The framework will be based on the Q-graph approach, and the Q-Graph approach is stochastic, so will the proposed framework behave differently in different runs when it is exposed to the same inputs? That is, is the new approach functionally stable?

**RQ-2:** Is the new framework more efficient and effective than the state-of-the-art algorithms for generating resetting and state-verification sequences?

(a) How efficient is the proposed approach, regarding memory and time requirement, when compared to the state-of-the-art algorithms?

(b) How effective is the proposed approach in terms of the sizes of the resetting and state-verification sequences compared to the sequences generated by other existing algorithms?

To assess the proposed framework and study the questions, we performed experimental evaluations on complete, partial, and timed systems. We used a number of synthetic models as well as benchmark models obtained from [41]. We also use an Engine Status Manager (ESM) model learnt from a printer controller: a partial model that has 77 inputs and 3410 states.

### 1.2 Contributions

We summarise the list of contributions as follows.

1) We improved and re-structured our Q-Graph learning algorithm to derive RSs and PDSs from complete and partial FSMs and tFSMs. Improvements (with respect to the earlier conference publication [40]) are summarised in Table 1;

<table>
<thead>
<tr>
<th>Seq. Type</th>
<th>FSM type</th>
<th>Approach in [40]</th>
<th>Proposed approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>Complete FSMs</td>
<td>✔</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Partial FSMs</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td></td>
<td>Timed FSMs</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>PDS</td>
<td>Complete FSMs</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Partial FSMs</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Timed FSMs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 1**: Capability comparison for the proposed framework
are similar to RSs: both take a system to a given state irrespective of the state in which it is applied. There are two important differences between the work regarding SSs previously reported and the work regarding RSs described in this paper. First, in this paper we devise a framework that can process both timed and untimed models; previously only untimed models were used. Second, we used a wider range of subjects in the experimental evaluation.

1.3 Organisation of the Paper
The paper is structured as follows. First, we describe related work in Section 2. Next, in Section 3, we provide the notation and definitions. Following this, Section 4 introduces the Q-Graph framework. We describe the experiments and results for un-timed FSMs in Section 5. Afterwards, we demonstrate how the Q-Graph framework can derive sequences from timed FSMs in Section 6. Following this, in Section 7, we discuss the Research Questions. Finally, in Section 8, we provide future directions and conclusions.

2 RELATED WORK
Existing model-based test generation approaches are tailored applications for a specific task and model class. For example, an algorithm for deriving RSs from complete FSMs cannot derive RSs from partial FSMs. An algorithm to derive RS or PDS from partial FSMs can be used to derive RS or PDS from complete FSMs with reduced performance: such an algorithm executes additional checks that are unnecessary for complete FSMs. Furthermore, an algorithm to generate PDSs from partial FSMs cannot derive RS from complete FSMs as they are different problems. However, we show that the proposed framework can be used to derive RS and PDSs from partial and complete FSMs and tFSM. The only change to be made to the framework is to alter the reward function.

In this section, we first review the related work associated with RSs and PDSs. This is then followed by a review of test generation using reinforcement learning.

2.1 Resetting sequences
There is a rich body of literature relating to the computational complexity of algorithms for RSs. To begin with, generating or checking the existence of an RS from a complete FSM is polynomial time solvable [42]. However, checking the existence of an RS of length \( k \) from such machines is NP-Hard [42]. Furthermore, checking the existence of an RS is PSPACE-Complete if the FSM is partial [43].

Much of the work regarding RSs aims to devise algorithms that can produce shorter RSs faster. Depending on the type of the underlying FSM, different approaches have been produced, and we separate these algorithms based on the underlying FSM, complete and partial and FSMs having timeouts.

Regarding complete FSMs, one of the fastest methods is the Greedy Method, which has an \( O(n^3 + in^2) \) time complexity, where \( n \) denotes the number of states and \( i \) denotes the number of inputs of the FSM. The initial step of the Greedy Method is the construction of a product-automaton, a data structure that requires \( O(n^2) \) space. Following this method, different approaches/heuristics to derive short RSs have been proposed [44], [45], [46], [47], [48], [49], [50], [51]. However, the tight bound \( O(n^3) \) prevents all the variations of this method from processing large models. In contrast, our approach does not require the construction of the product automaton.

Another algorithm for calculating RSs has been presented by Gürünün et al. [52]. This algorithm uses Answer Set Programming to generate RSs from complete FSMs. However, encoding the system and the RS generation problem into a Conjunctive Normal Form formulation is not scaling to large systems with many states [52].

In order to scale, researchers developed algorithms that run on general-purpose graphics processing units (GPUs) [17], [18]. Even though this is an active research direction and the results are encouraging, these methods cannot handle partial systems and are difficult to program. Their performance solely relies on the underlying GPU hardware.

For partial FSMs, the standard algorithm uses brute force search [30], [53]. The only scalable improvement [12] checks all input sequences that are not longer than some upper bound. Experimental results show that the algorithm can process FSMs with 16,000,000 states and 10 inputs using a relatively powerful GPU card and CPU. However, the algorithm’s performance drastically drops as the number of inputs increases. Moreover, as this algorithm runs on a GPU, it suffers from the abovementioned issues (equipment and expertise requirements).

For tFSMs, scalability is adversely affected by the fact that the process of deriving RSs from complete (or partial) tFSMs involves a step that creates a complete (or partial) FSM with many more states than the corresponding tFSM. Our approach can derive RSs directly from the tFSM models without creating partial (or complete) FSMs [33].

2.2 Preset Distinguishing Sequences
The problem of checking the existence of a PDS is PSPACE-complete even for complete FSMs [15], and to our knowledge, the existing methods to derive PDSs involve a brute-force approach [30]. A GPU-based brute-force PDS generation algorithm has been proposed for FSMs, and it is worth noting that this algorithm also works for non-deterministic FSMs [55]. Moreover, an SAT-based PDS generation algorithm for complete FSMs has been given [56]. However, the time required to translate the system and the PDS generation problem into a Conjunctive Normal Form is high, and the method cannot process large FSMs.

Similar to RSs, it is important to note that no published method is processing tFSMs to generate PDSs. However, again it is possible to pre-process the tFSM and map a tFSM to an FSM and then use standard algorithms. Due to the cost of the pre-processing step, this method suffers from poor efficiency.

2.3 Test generation using Reinforcement Learning
Motivated by recent breakthroughs in Reinforcement Learning (RL), the work presented in this paper aims to use RL as
the basis for novel algorithms that efficiently find short RSs and PDSs. RL is a family of machine learning algorithms and is becoming an essential tool in computational intelligence [57]. In RL, computers (agents) make their own choices (take actions) in a given environment without having prior information or labelled data [58].

Recently, the use of RL has received attention in several areas of testing, including Android testing [59], mutation testing [60], online testing [61], and security testing [62], [63]. Except for our past work [40] no approach has been proposed for FSM-based testing. We now briefly describe a few of these lines of work.

In Android testing, the focus has been on automatically generating test cases to improve code coverage [59], while the mutation testing work has used RL to predict whether a given test suite kills a mutant without incurring the cost of executing the mutant [60]. In online testing, test inputs are chosen during test execution, and RL has been used to address the problem of optimising the choices made to reduce test costs [61]. RL has been used to learn a behaviour model of the system under test to aid risk-based testing [65].

Three RL algorithms have been proposed and embedded in EvoSuiteFIT [66] to support hyperheuristic search-based test generation algorithms [67]. Within security testing, researchers have developed an RL-based testing algorithm that trains dishonest agents to reveal dangerous behaviours of autonomous cyber-physical systems [64]. In addition, an RL-based test generation technique has been devised to increase hardware Trojan detection accuracy [62], [63].

3 Preliminaries

3.1 Finite State Machines

A Finite State Machine (FSM) is an abstract machine that has a finite set of states (S), one of which is designated as the initial state (s₀ ∈ S), finite set of inputs (X) and a finite set of outputs (Y). When an input x ∈ X is applied, when the FSM is in state s, the FSM produces an output y ∈ Y and changes its current state to some s’ ∈ S. We summarised the terminology we used for FSMS in Table 2.

In Figure 1, we give a complete FSM M₁ that models a ShiftRegister circuit [41]. With respect to FSM M₁, if a tester applies input x₁ when the FSM is in state s₀, then the FSM will produce output y₀ and change its state to s₁. Given set A, we use A+ to denote the set of sequences of zero or more elements from A, and ε is used to denote an empty sequence.

In this paper, we consider deterministic FSMS and simply call these FSM. An FSM’s behaviour is represented by next state δ and output λ functions which define transitions. A transition τ has a starting state start(τ) ∈ S, an input symbol in(τ) ∈ X, an ending state end(τ) = δ(s, x), and an output symbol out(τ) = λ(s, x). Thus, we can represent an FSM using the tuple M = (S, X, Y, δ, λ, s₀). If δ and λ are defined on all pairs in S × X then M is a complete FSM; otherwise M is a partial FSM. If M is complete then δ has type S × X → S and λ has type S × X → Y. That is, δ is a function that maps (state × input) pairs to states, and λ is a function that maps (state × input) pairs to output symbols. Note we used ε as an input to indicate the cases in which FSM does not receive input. If M is partial, we use ε as the output of δ and λ whenever they are applied to a pair from S × X on which they are not defined. Thus, for partial FSMS we have functions types δ : S × X → S ∪ {ε} and λ : S × X → Y ∪ {ε} ; if x is an undefined input for s then we have δ(s, x) = ε and λ(s, x) = ε. Similarly, if x = ε, then we have δ(s, ε) = s and λ(s, ε) = ε.

An input sequence ŷ ∈ X* is a sequence of inputs, i.e., ŷ = x₁ x₂ . . . xₖ. If an input sequence ŷ is applied to a complete FSM M in state s, the FSM performs a walk defined by a sequence of transitions τ₀ τ₁ τ₂ . . . τₖ. These are consecutive transitions: for all 1 ≤ i < k, we have that end(τᵢ) = start(τᵢ₊₁). If one concatenates the input symbols of these transitions, then one obtains the trace x₁ x₂ . . . xₖ.

Suppose that ŷ is a finite input sequence from X*; if the FSM M is complete, then for every state s of M, there is a walk starting from s that has trace ŷ. However, if M is partial, then such a walk might not exist starting from s due to the missing transitions. For partial FSMS, the FSM performs a walk if ŷ is a defined input sequence for s. An input sequence ŷ is defined in state s, if for all non-empty prefixes ŷ’ x of ŷ, x is defined in the state reached from s with ŷ’.

We extend δ and λ to input sequences in the following way: let ŷ = ŷx be an input sequence and s ∈ S be a state of FSM M, then δ(δ(s, x), x’) and λ(λ(s, x’)) = λ(λ(s, x), λ(δ(s, x), x’)). Note that for a state s and input sequence ŷ, the functions returns symbol ε if ŷ is not defined for s, and returns s if ŷ = ε. Furthermore, we extend δ to a set of states S’ ⊆ S as follows: given an input x, output y and state set S’ ⊆ S then δ(δ(S’, x)) denotes the set of states one can reach from a state in S’ with a transition with input x and output y.

A transition τ has a starting state start(τ) ∈ S, an input symbol in(τ) ∈ X, an ending state end(τ) = δ(s, x), and an output symbol out(τ) = λ(s, x). Thus, we can represent an FSM using the tuple M = (S, X, Y, δ, λ, s₀). If δ and λ are defined on all pairs in S × X then M is a complete FSM; otherwise M is a partial FSM. If M is complete then δ has type S × X → S and λ has type S × X → Y. That is, δ is a function that maps (state × input) pairs to states, and λ is a function that maps (state × input) pairs to output symbols. Note we used ε as an input to indicate the cases in which FSM does not receive input. If M is partial, we use ε as the output of δ and λ whenever they are applied to a pair from S × X on which they are not defined. Thus, for partial FSMS we have functions types δ : S × X → S ∪ {ε} and λ : S × X → Y ∪ {ε} ; if x is an undefined input for s then we have δ(s, x) = ε and λ(s, x) = ε. Similarly, if x = ε, then we have δ(s, ε) = s and λ(s, ε) = ε.

An input sequence ŷ ∈ X* is a sequence of inputs, i.e., ŷ = x₁ x₂ . . . xₖ. If an input sequence ŷ is applied to a complete FSM M in state s, the FSM performs a walk defined by a sequence of transitions τ₀ τ₁ τ₂ . . . τₖ. These are consecutive transitions: for all 1 ≤ i < k, we have that end(τᵢ) = start(τᵢ₊₁). If one concatenates the input symbols of these transitions, then one obtains the trace x₁ x₂ . . . xₖ.

Suppose that ŷ is a finite input sequence from X*; if the FSM M is complete, then for every state s of M, there is a walk starting from s that has trace ŷ. However, if M is partial, then such a walk might not exist starting from s due to the missing transitions. For partial FSMS, the FSM performs a walk if ŷ is a defined input sequence for s. An input sequence ŷ is defined in state s, if for all non-empty prefixes ŷ’ x of ŷ, x is defined in the state reached from s with ŷ’.

We extend δ and λ to input sequences in the following way: let ŷ = ŷx be an input sequence and s ∈ S be a state of FSM M, then δ(δ(s, x), x’) and λ(λ(s, x’)) = λ(λ(s, x), λ(δ(s, x), x’)). Note that for a state s and input sequence ŷ, the functions returns symbol ε if ŷ is not defined for s, and returns s if ŷ = ε. Furthermore, we extend δ to a set of states S’ ⊆ S as follows: given an input x, output y and state set S’ ⊆ S then δ(δ(S’, x)) denotes the set of states one can reach from a state in S’ with a transition with input x and output y. We therefore define δ(S’, x) to be ε if there is some state s’ ∈ S’ such that x is undefined in s and otherwise δ(S’, x) = {s’ ∈ S’ ∣ δ(s, x) = s’ ∧ λ(s, x) = y}.

Let Ŝ = {S’, S’, . . .} denote a set of sets of states, i.e., for all S’ in Ŝ we have that S’ ⊆ S. We use δ(Ŝ, x) to denote the outcome of applying an input to a set of sets of states as defined below:

\[
\hat{\delta}(\hat{S}, x) = \begin{cases} 
\hat{\delta}(\hat{S}, x), & \text{if } \exists S’ \in \hat{S} \text{ s.t. } x \text{ is undefined for } S’ \\
0, & \text{if } \exists S’ \in \hat{S} \text{ s.t. } x \text{ is merging for } S’ \\
\{\delta(S’, x) \mid \exists s \in S’ \cdot y = \lambda(s, x), S’ \in \hat{S}\}, & \text{otherwise.}
\end{cases}
\]

Where x is merging for set S’ of states, if x is defined for S’ and there exist distinct s and s’ in S’ such that δ(s, x) = 2. It is usual to extend δ and λ to a set of sets of states and also to input sequences because it helps simplify other definitions.

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>S, S₀</td>
<td>Set of inputs, set of outputs, input, output, input sequence, output sequence.</td>
</tr>
<tr>
<td>δ, λ</td>
<td>Transition function for a set of sets of states, transition function, output function.</td>
</tr>
<tr>
<td>pow(·)</td>
<td>Power set of (·).</td>
</tr>
</tbody>
</table>

TABLE 2 List of symbols regarding FSMS
A resetting sequence (RS) of an FSM $M$ is an input sequence $\bar{x}$ that takes $M$ to a specific state regardless of the state where $M$ was before the application of $\bar{x}$. For example, consider the FSM $M_1$. If a tester applies input sequence $\bar{x} = x_1 x_1 x_1 x_1$, $M_1$ will reach state $s_7$ regardless of the initial state.

**Definition 3.1.** An input sequence $\bar{x}$ is a resetting sequence for FSM $M = (S, X, Y, \delta, \lambda, s_0)$ if and only if $\bar{x}$ is a defined input sequence for $S$ and $|\delta(S, \bar{x})| = 1$.

### 3.2 Resetting sequences

A resetting sequence (RS) of an FSM $M$ is an input sequence $\bar{x}$ that takes $M$ to a specific state regardless of the state where $M$ was before the application of $\bar{x}$. For example, consider the FSM $M_1$. If a tester applies input sequence $\bar{x} = x_1 x_1 x_1 x_1$, $M_1$ will reach state $s_7$ regardless of the initial state.

**Definition 3.1.** An input sequence $\bar{x}$ is a resetting sequence for FSM $M = (S, X, Y, \delta, \lambda, s_0)$ if and only if $\bar{x}$ is a defined input sequence for $S$ and $|\delta(S, \bar{x})| = 1$.

### 3.3 Preset Distinguishing Sequences

A preset distinguishing sequence for an FSM $M$ is an input sequence that leads to different output sequences when applied in the different states of $M$; formally.

**Definition 3.2.** An input sequence $\bar{x}$ is a PDS for FSM $M = (S, X, Y, \delta, \lambda, s_0)$ if $\bar{x}$ is a defined input sequence for $S$ and there do not exist any $s, s' \in S$ such that $s \neq s'$ and $\lambda(s, \bar{x}) = \lambda(s', \bar{x})$.

Observe that input sequence $\bar{x} = x_0 x_1 x_1 x_1$ is a PDS for $M_1$. For example, if a tester applies $\bar{x}$ when the FSM is in $s_0$, the FSM will produce $y_0 y_0 y_0 y_0$, but it will produce $y_1 y_1 y_0 y_0$ if $M_1$ was in state $s_8$. No pair of states will produce the same output sequence, and we can distinguish the states by using $\bar{x}$.

### 3.4 Q-Learning environment

Q-learning algorithms operate on a Markov Decision Process (MDP) [39]. An MDP is defined as a tuple $P = (Z, A, P, R)$, where $Z$ is a set of states, $A$ is a set of actions, $P$ is the probability function in the form of $P(k' | k, a)$, that is for each action $a \in A$ and state $k \in Z$, it defines the probability of moving to state $k' \in Z$ if action $a$ occurs in state $k$. $R$ is the immediate reward function in the form of $R : Z \times A \times Z \rightarrow \mathbb{R}$, i.e., it returns the reward received after transitioning from one state to another state with action $a$ where $\mathbb{R}$ is the set of real numbers.

Q-learning is a value-based reinforcement learning method which is used to find the optimal policy when state transition probabilities $P$ are unknown for a given MDP $P = (Z, A, P, R)$. Instead of estimating these unknown probabilities, the method uses a value function $Q(K, a)$ [68]. Let $K'$ be a state reached from $K$ using action $a$. The value function $Q(K, a)$ is recursively defined as:

$$Q(K, a) = \begin{cases} 
Q(K, a) + \alpha \{ R(K, a) + \gamma \arg \max_{a'} Q(K', a') \} & \text{if } K = \text{current state, and an action } a \text{ is executed} \\
\text{no change, otherwise,} & \end{cases}$$

where $K \in Z$, $a \in A$, and $Q(K, a)$ is the value of applying action $a$ at state $K$, $a'$ and $K'$ are the next action and next states, respectively, $R(K, a)$ is the immediate reward received after applying action $a$ at state $K$, $\alpha$ is the learning rate, and $\gamma$ is the future reward discount factor. $\alpha$ and $\gamma$ are values within the range $[0, 1]$.

The Q-learning algorithm asymptotically reconstructs the true expected discounted reward [68] and, as a result, works towards recovering the optimal policy. In this respect, policy selection based on Q-learning can be viewed as an off-policy temporal difference control algorithm which asymptotically approximates the optimal policy [39].

### 4 Q-GRAPHS

A Q-learning algorithm requires a function that calculates the quality of a state–action combination. The naive implementation of the Q-learning algorithm relies on a Q-table, which holds the Q-values for each of the states of the underlying environment, i.e., MDP $P$ [39]. Even though the use of a table guarantees that the learning will converge to an optimum policy, the size of the table usually becomes a bottleneck for systems employing large state-action spaces, and there are a number of methods to avoid this bottleneck [69], [70]. For example, consider an environment characterised by
64 binary-valued parameters and 4 actions. To represent this environment in a tabular environment, one needs to build a $2^{32}$ by 4 table leading to 68.7GB RAM. One promising solution is to use deep neural networks (DNNs) as function approximators for the optimum policy. Deep Q-Learning algorithms generally yield good results \[38\]. However, using DNNs requires a solid understanding of the DNNs, and they make the overall system opaque, i.e., hard to trace errors and reveal reasoning followed by the agent \[71\]. We show that the Q-Graph framework can reduce the memory requirement by more than 99.99% and can find optimum results.

4.1 The generic Q-Graph algorithm

In practice, in tabular Q-learning, the agent usually only needs to visit some of the state-action pairs to compute an optimum policy; therefore, using a table will lead to a waste of memory. Moreover, we also note that the rules regulating the underlying environment dictate state changes as the agent applies actions. That is, an agent will move to another state $K'$ after applying action $a$ from state $K$ and this state transition information, which we call the learning flow, is not kept in the tabular setting. A learning flow is the agent's node visit pattern within the Q-Graph while learning. The learning flow is essential to investigate the agent's behaviour during learning. We also observed that this state-action-state transition, a.k.a. learning flow, can naturally be encapsulated as a graph. Based on these observations, we consider state information of the environment $K$ as a graph node that will keep the quality values of the actions for that state. The edges preserve the learning flow, which is impossible in tabular Q learning. We illustrate this new framework in Figure 2 and formally define Q-Graphs as follows.

**Definition 4.1.** A Q-graph $Q = (N, E)$ has a finite set of nodes (Q-nodes) $N$ and edges $E$. Each node $n \in N$ is associated with a state of the environment $K$ denoted as $n_K$ and a $Q$-value ($Q(K, a)$) for each action $a$ of $P$.

Intuitively, in the set of admissible edges in a Q-graph there exists an edge labelled with action $a$ from Q-node $n$ to Q-node $n'$ if and only if one can reach $n_K$ from $n_K$ using an action $a$. We summarised the new terminology we introduced for Q-learning in Table 3.

A generic algorithm for the Q-Graph framework is given in Algorithm 1. Note that the algorithm is similar to the generic tabular Q-learning algorithm \[39\]. The first step of this algorithm is to select random Q-values for each action of the initial node and create the initial node (Lines 1-4 of Algorithm 1). After initialising the Q-Graph, the algorithm enters a loop that repeats as long as there is an episode: a limit set by a programmer/domain expert. During each episode, the agent starts interacting with the environment from an initial dedicated node (Line 6 of Algorithm 1) and keeps interacting with the environment as long as the agent does not reach a terminal state ($n''$) for that episode (the While loop between Lines 7-18 of Algorithm 1). The properties characterise a terminal state depending on the context, i.e. a robot may reach a trap, a drone may reach a closed environment, etc. At each episode, the agent has to introduce missing adjacent nodes for the current node ($n_K$) it reaches (Lines 8-11 of Algorithm 1). This is then followed by selecting the action to be applied by using a strategy such as $\epsilon$-greedy\[4\], collecting the immediate reward from the environment, and updating the Q values (Lines 12-14 of Algorithm 1). Afterwards, depending on the reward and the next state, the algorithm either moves to the node that abstracts the new state or terminates the episode (Lines 15-18 of Algorithm 1).

A key observation here is that in the worst case, a Q-graph will keep information for all state-action pairs and it is guaranteed that the Q-Graph converges to an optimum policy if sufficiently many nodes are explored. In the experiments, we will show that the Q-Graph can reduce the space requirements by more than 99.9% and computes an optimum policy.

**Algorithm 1:** A generic Q-Graph algorithm.

1. Initialise a Q-node $n_0$ with the initial state $K_0$.
2. foreach $a \in A$ do
3. Assign a random value to $Q(K_0, a)$ of $n_0$.
4. $N \leftarrow N \cup \{n_0\}$.
5. while There is an episode do
6. $n' \leftarrow n_0$.
7. while Current episode do
8. foreach $a \in A$ do
9. if Node $n'$ with $n_K''$, does not exist in $N$ then
10. Introduce $n''$ to $N$ having random Q-values.
11. Introduce an edge from $n'$ to $n''$ labelled
12. Take an action $a \in A$ using policy derived from $Q$
13. Record reward and next state $K''$.
14. Update $n'$, and Q-values according to Q function.
15. else

3. This table would have 17,179,869,184 cells each having four 4-byte word (for holding the quality values).

4.2 Constructing resetting sequences using Q-Graphs

We represent the RS construction problem as an MDP $P$. An element $S'$ of the power set of $S$ ($S' \in pow(S)$) corresponds

4. In $\epsilon$-greedy, the agent guesses a value between 1 and 0, and if the value is less than some threshold, it picks an action randomly; otherwise, it picks the action suggested by the Q function.

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Z, A, K$</td>
<td>Set of MDP states, Set of actions, MDP state.</td>
</tr>
<tr>
<td>$P, R$</td>
<td>Transition probability function, Immediate Reward Function.</td>
</tr>
<tr>
<td>$Q(K, a)$</td>
<td>The quality value of taking action $a$ at MDP state $K$.</td>
</tr>
<tr>
<td>$Q, N, E, n, n_K$</td>
<td>Q graph, set of Q nodes, set of edges, a Q node, MDP state associated to Q node. (MDP state-set of FSM states) conversion function. (MDP action-FSM input) conversion function.</td>
</tr>
</tbody>
</table>

**Table 3** List of symbols regarding Q-learning.
to a state \((K' \in Z)\) of the MDP \(P\) and each input \(x \in X \) of \(\text{FSM} \ M\) is an action \(a \in A \) 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 FSM \(M\) to the corresponding action \(a \in A\) of the MDP \(P\) and vice versa, and (ii) \((st())\) maps a set of states \(S' \in \text{pow}(S)\) of the FSM \(M\) to the corresponding state \(K' \in Z\) of the MDP \(P\) and vice versa. In this setting, the Q-learning agent’s task is to reach a state \(K\) such that \(|\delta(st(K))| = 1\). In other words, the agent will learn a sequence of inputs in the form of actions that resets \(M\). For example, in Figure 3, we observe that rewards are generated from the MDP state \(K\) due to two actions. One of them does not reduce the number of states \(K\), so the reward is \(-1\). Another action, however, merges a state and causes the heuristic function to take place (which returns a reward value of 33.3).

For each \(K\) and \(a\), we let

\[
P_{RS}(K', a) = \begin{cases} 1, \text{ iff } \delta(st(K), i(a)) = st(K'), \\ 0, \text{ otherwise.} \end{cases}
\]

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

The immediate rewards \((R_{RS}(K, a))\) are computed as follows in which \(K'\) is the next MDP state such that \(\delta(st(K), i(a)) = st(K')\).

\[
R_{RS}(K, a) = \begin{cases} -1, & \text{if } |st(K)| = |st(K')|, \\ \text{NAN,} & \text{if } i(a) \text{ is undefined for } st(K) \\ 100 * \phi & \text{else } (\phi = |st(K)| - |st(K')|). \end{cases}
\]

The above formulation introduces a heuristic that helps to break ties when we have a set of resetting inputs. The heuristic step considers the number of reset states and promotes the input, causing more states to merge. Note that when \(|S'\|\) equals \(|S|\), the reward is \(-1\); this is a step to prevent the agent from introducing redundant inputs to RS. Moreover, with an immediate reward of \(\text{NAN}\), the proposed algorithm can derive RSs from partial systems without worrying about constructing RSs with undefined inputs. However, the algorithm may generate longer RSs when the underlying system is partial. This is because the length of the shortest RS for a complete system is bounded by \(O(n^2)\) [12], and for a partial system, the bound is \(O(n^2 + 4n^3)\) [72].

4.3 Constructing preset distinguishing sequences using Q-Graphs

For a given FSM \(M\), searching a PDS often starts from a set \((\hat{S})\) that contains the set of states of the underlying FSM \(M\), i.e. \(\hat{S} = \{S\}\) and ends in a set \(\hat{S}'\) such that \(\hat{S}' = \{\{s_0\}, \{s_1\}, \ldots, \{s_{n-1}\}\}\), where \(n\) is the number of elements of \(S\). The search uses inputs \(X\) of the underlying FSM \(M\) and the \(\hat{\delta}\) function to explore sets of sets of states. For example, assume \(X = \{x, x'\}\). The search will evaluate the reached sets \(\hat{\delta}(S, x)\) and \(\hat{\delta}(S, x')\) from \(\hat{S}\) and selects one input (say \(x'\)). After this, it appends \(x'\) to the PDS under construction \((\hat{x})\) and forms a set \(\hat{S}' = \hat{\delta}(S, x')\) and repeats the steps given above, this time using \(\hat{S}'\). If neither of the inputs leads to a solution, then the search often backtracks. This process continues when the search finds a set \(\hat{S}'\). Therefore if a PDS \(\hat{x} = x, x', x''\ldots\) is found, then \(\hat{\delta}(\hat{S}(x), x, x'\ldots) = \hat{S}'\).

To simulate this search, as in the case of the RS formulation, in the PDS formulation, we consider each input \(x \in X\) as an action \(a \in A\) of \(P\). However, a state \(K \in Z\) of the MDP \(P\) is associated with a set of sets of states \((\hat{S})\) that is a subset of \(\text{power}(S)\). Since FSM \(M\) has \(n\) states, the number of elements of an \(S\) associated with a \(K \in Z\) of the MDP \(P\) varies from 1 where \(\hat{S} = \{S\}\) to \(n\) where \(\hat{S}' = \{\{s_0\}, \{s_1\}, \ldots, \{s_n\}\}\). So while generating a PDS, a Q-learning agent’s task is to find a sequence of actions (inputs) that reaches an MDP state \(K'\) such that \(|\hat{\delta}(\hat{S}(K))| = n\) from an MDP state \(K\) such that \(|\hat{\delta}(\hat{S}(K))| = 1\) by using the \(\delta\) function of the underlying FSM. Here, \(\hat{\delta}(\hat{S})\) is a one-to-one and onto function, which receives an MDP state \(K\) and returns a set of sets of states \(\hat{S}\) and vice versa. According to this, using \(i()\) and \(st()\) we define \(P\) of \(P\) as follows; for each \(K\) and \(a\), we let

\[
P_{PDS}(K', a) = \begin{cases} 1, \text{ iff } \hat{\delta}(\hat{\delta}(\hat{S}(K), i(a)) = st(K'), \\ 0, \text{ otherwise.} \end{cases}
\]

That is, in \(P\) there exists a transition from a given state \(K\) to another state \(K'\) labelled with action \(a\) if and only if corresponding input \(x\) is defined for the states in \(\hat{S}\) and \(\hat{\delta}(\hat{S}(K), x) = \hat{S}'\).

The immediate reward function \((R_{PDS}(K, a))\) penalises undefined and merging inputs and inputs that do not increase the number of sets in \(\hat{\delta}(\hat{S})\). However, it rewards inputs that increase the number of sets in \(\hat{\delta}(\hat{S})\) evenly. To achieve this, we consider the weight of input \(x \in X\) \((\kappa(x))\) by using the following function. Let \(S'\) be a set of states having more than one element due to the application of input \(x\), then \(\kappa(x) = \sum_{S', S'' \in S'}|S'| - |S''|\).

This heuristic is effective in producing short sequences [73]. The intuition behind the heuristic is that large differences in the number of elements indicate that some sets are smaller than others. However, the number of inputs required to distinguish states within a set \(S'\) is bounded by \((\binom{n}{2})\) where \(\eta = |S'| \) and \(n = |S|\). So the immediate reward for the PDS problem is computed as follows in which \(\hat{K'}\) is the next MDP state such that \(\delta(st(K), i(a)) = st(K')\).

\[
R_{PDS}(K, a) = \begin{cases} -1, & \text{if } |st(K)| = |st(K')|, \\ \text{NAN,} & \text{if } i(a) \text{ is undefined or merging for st}(K) \\ |S| - \kappa(i(a)) & \text{else}. \end{cases}
\]

In Figure 4, we illustrate the reward function in action. We see that the action that partitions the set of states in MDP state \(K\) gets reward value 3, while the other action gets \(-1\).
4.4 Accuracy and space requirement for Ψ-Graph

In the worst case, the agent will visit all the state-action pairs. Therefore, the number of nodes is bounded by the set $pow(S)$, where $S$ is the set of states of the underlying FSM.

Corollary 1. Let $M$ be an FSM having $n$ states. Then to construct an RS (or a PDS), the space complexity of the Ψ-Graph framework is bounded above by $O(2^n)$.

Finally, note that the set $pow(S)$ defines a limit for the MDP $P$; hence $P$ is finite.

Corollary 2. Let $M$ be an FSM; the MDP $P$ constructed from $M$ for constructing an RS is a finite MDP.

Corollary 3. Let $M$ be an FSM; the MDP $P$ constructed from $M$ for constructing a PDS is a finite MDP.

This implies that Ψ-Graph can find optimum policies for RS and PDS problems.

Corollary 4. Let $M$ be an FSM having $n$ states. Then the Ψ-Graph framework is guaranteed to construct an RS iff $M$ has one.

Corollary 5. Let $M$ be an FSM having $n$ states. Then the Ψ-Graph framework is guaranteed to construct a PDS iff $M$ has one.

Algorithm 5: An FSM generator algorithm.

```
input : The number of FSMs, states, and input/output alphabets are positive integers and are noted as $l$, $n$, $i$, and $j$, respectively. The connectedness property is noted as $c \in \{0, 1\}$ where $0$ for connected, $1$ for initially connected FSMs. Rate of partial transitions of FSMs is given in the range $0 \leq p < 100$.
output: $l$ number of FSMs having properties dictated by inputs.
1 Create $S$, $X$, $Y$ sets having $n$, $i$, and $j$ items
2 while FSMs are not created do
3     foreach $s \in S$ and $x \in X$ do
4         Introduce a transition from $s$ to a randomly chosen state $s'$ with randomly chosen output symbol $y \in Y$ and set $d(s, x) = s'$ and $\lambda(s, x) = y$
5     if $p > 0$ then
6         Randomly select $(n \times x)/100$ transitions and remove them.
7     if $c=0$ and FSM is not connected then
8         Goto Line 2
9     if $c=1$ and FSM is not initially connected then
10        Goto Line 2
11     if FSM has no PDS and/or RS then
12        Goto Line 2
13        Record FSM.
```

5 Experiments

This section reports experimental findings with our test subjects and benchmark algorithms. The experiment subjects in this section are unimodal. Experiments regarding timed FSMs are given in Section 6. We provide the source code and the synthetic machines in the following repository https://zenodo.org/record/8043032.

5.1 Experiment setting

5.1.1 Test subjects and the software and the hardware

In the experiments, we used two classes of FSMs: synthetic FSMs and real FSMs. Real FSMs were retrieved from the embedded systems industry, modelling real circuit systems. The synthetic FSMs were constructed using a tool (FSM Generator) used in similar studies in the literature [12], [54], [56], [74], [75]. We provide the steps taken by FSM Generator in Algorithm 5.

For this work, we created two types of synthetic FSMs; complete FSMs and partial FSMs. So as to provide as many variations of FSM models as possible while creating them, we fed Algorithm 5 with parameters $n$, $i$, $j$, $p$, and $l$ using the following values. The $n$, $i$ values are set from sets \{32, 64, 128, \ldots, 16384\} and \{2, 3, 4, 5\}, respectively and also we set $i = j$. In order to reduce the time required by tests, we take $p = 10$ as a rate for missing transitions while creating partial FSMs. We also set the number of FSMs ($l$) as 100. For partial and complete FSMs and each different ($i$, $j$) pair, we created 100 FSMs, and in total, we used 8,000 FSMs.

The second class of FSMs consists of 59 specifications of circuit behaviour [41]. The circuit benchmarks have been used recently for testing [75]. The models were originally Kiss2 files having various properties. We used the processed versions, which are given as DOT files, and all are deterministic and complete [41]. All the FSMs are complete, and we summarised the FSMs in Table 4. Moreover, we also added an FSM to the second class, which is the Engine Status Manager (ESM), that manages the status of the engine in Océ printers and copiers (a subsidiary of Canon). This example is chosen because its structure and behaviour are representative of embedded control software [11]. The ESM model was learned by interacting with the actual system using another piece of software, LearnLib [76] 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.

For experiments, we created two versions for the ESM specification: \(v_1\) and \(v_2\). \(v_1\) was created by completing the missing transitions to obtain a completely specified model, using a widely applied method [3], [75]. To complete the missing transitions in the ESM model, an error-state was introduced, where for each state \(s\) and each missing transition labelled by input \(x\), a transition was introduced from \(s\) to the error-state with input \(x\) and output \(e\). The second version of ESM was the original partial version. According to [40], the ESM model does not have an RS because all the transitions leaving specific state pairs also end in those pairs, namely \((s_{524}, s_{721})\), \((s_{70}, s_{71})\), \((s_{304}, s_{3088})\), \((s_{344}, s_{2933})\), and \((s_{1080}, s_{2358})\). To use the FSM in the experiments, transitions labelled with a common input symbol \(T_2)1.1\) for each of these 10 states were modified to merge at state \(s_{524}\). We applied Wilcoxon non-parametric paired tests, and Cohen’s d tests on the experiment result [77], [78], we provided details of these tests in Section 5.1.4.

We used a Microsoft Windows 11 operated computer having 32GB RAM with 11th Gen Intel(R) Core(TM) i7 – 11800H running at 2.30GHz. We implemented all the algorithms in C++ under Microsoft Visual Studio 19. We implemented the algorithm in [12] under CUDA 12 runtime environment and used NVIDIA T1200 Laptop GPU with 4GB RAM with 1024 CUDA cores. We use the R tool to generate plots and conduct analysis [79]. We provided the R tool scripts for producing the illustrations in https://github.com/urazc/Q_Graph.

5.1.2 Benchmark algorithms

We use the notation Q(\(\beta\)) to refer to our approach, where variable \(\beta\) denotes the corresponding reward function, i.e., Q(RS) denotes the reward function for reset sequence generation. We used three different algorithms to assess the effectiveness of our method. Firstly, we used the Greedy method (GREEDY) to build an RS from a complete machine, which is widely recognised and the fastest sequential algorithm, requiring \(O(n^3 + in^2)\) computation steps [42]. We provided this algorithm in Algorithm 2. The Greedy algorithm constructs a product automaton, which merges each pair of states and takes quadratic space and time in proportion to the number of states of the FSM. Secondly, we utilised the Parallel Brute-Force (PBF) RS generation algorithm to generate RS from partial FSMs, which is currently the fastest known algorithm for this purpose [12], [18]. We provided the steps of this algorithm in Algorithm 3. Thirdly, we implemented the Successor-Tree (ST) algorithm given in [30] to create PDs from FSMs, which is a classic algorithm described in detail in Algorithm 4. During experiments we could not use the tabular Q-learning algorithm due to space limitations. For example for the smallest FSM subject, which has 32 states and two inputs, the tabular Q-learning framework would require a \(2^{32}\) by 2 table leading to 8,559,349,592 entries each holding two (for holding Q-values) 4 byte words summing up a 34GB memory space which we did not have during the experiments. We provide more details in Section 7.

5.1.3 Collected data

While algorithms were processing the underlying FSMs, we collected computation-related data. First, to answer RQ-1 and assess the accuracy and the stability of the Q-Graph algorithm, we noted the collected rewards and the Q-nodes and edges information of the Q-Graph. Second, to answer RQ-2, we collected the used resources, such as the execution time (in milliseconds) and used memory (in MBs). While measuring the time used by the Q-learning algorithm, we collected the sum of the training time and inference time and provided results based on this sum. Finally, to compare the lengths of sequences and to conduct a validity analysis, we kept the constructed sequences and the lengths of these sequences. We provide answers to RQs and discuss their implications in Section 7, after presenting the results of experiments.

5.1.4 Statistical analysis

In order to investigate our results, we used the paired Wilcoxon statistical analysis test using a .05 significance level to check whether there was a statistically significant difference between the results [77]. The null hypothesis examines whether the distributions are identical. The null hypothesis is accepted if the p-value exceeds 0.05. This test relies on three assumptions regarding the populations being compared: (i) the samples are independent, (ii) the populations have equal variance or spread, and (iii) the populations do not follow a known distribution, making it a non-parametric test. The Wilcoxon test identifies the existence of a statistical difference. However, it does not say
how strong the difference is. We applied Cohen’s d statistical test [78] to realise this. Cohen’s d test measures the level of strength of the difference between two results. The result of Cohen’s d test is classified as strong difference if the value is larger than 0.5, medium difference if the value is between 0.2 and 0.5 and if it is less than 0.2, the difference is considered as weak.

5.2 Results on PDSs derived from complete FSMs

5.2.1 Time & memory

Figure 5 summarises the time requirements of the algorithms. The results reveal that the proposed algorithm (Q(PDS)) significantly reduces the time required to compute PDSs, with a 100-fold average improvement. It is worth noting that the speed gain is approximately 4-fold on average when the number of states is fewer than 1024, and there are two inputs, while in other settings, the average speed increase is 250-fold. The results of statistical analysis, regarding time, are given in Tables 5 and Table 6. We can see a solid difference in Wilcoxon test results when \( n \geq 100 \). Cohen’s d test also indicates a strong difference in the time requirements of the algorithms.

The memory requirement comparison (Figure 6) demonstrate that Q(PDS) reduces the memory usage required to generate PDSs from complete FSMs by roughly 66 times on average and is 100 times in maximum (Figure 6). With respect to memory usage, we see that the Wilcoxon significance test declares a difference between the results on memory requirements of Q(PDS) and ST (Table 5), results are also supported with Cohen’s d test given in Table 6.

5.2.2 Length of sequences

Regarding the lengths of PDSs (Figure 7), it is observed that the ST algorithm produces shorter PDSs than the Q(PDS) method (with an average reduction of 12%). This is expected as the ST algorithm is a brute-force algorithm, which computes one of the shortest PDSs for the underlying FSM (Figure 7). The paired Wilcoxon hypothesis test results are given in Table 5. Results suggest that the experiment’s findings are statistically different. However, the test accepts the null hypothesis when \( n = 64 \) and \( i = 2 \) or 3 (result supported by Cohen’s test given in Table 6). Combining this result with Figure 5 we conclude that, in general, Q(PDS) generates longer sequences, but the lengths of PDSs for this subset of FSMs were similar.

5.3 Results on PDSs derived from partial FSMs

5.3.1 Time & memory

Experiment results with respect to time and memory collected during computing the partial FSMs are given in Figures 8 and 9, respectively.

5.4 Results on RSs derived from complete FSMs

5.4.1 Time & memory

The charts in Figures 11, 12, summarise the time and memory requirements of algorithms used to create RSs from

\[ \frac{(3600 - 240)}{240} \times 100 = \frac{(3360/240)}{100} = 1400\% \]
complete FSMs. The GREEDY algorithm could not handle FSMs with more than 512 states because it runs in cubic time (the timeout for these experiments was $10^3$ milliseconds). However, the proposed algorithm (Q(RS)) successfully generated RSs from FSMs with 16384 states and increased the scalability by 100%.

Specifically, considering the Figure 11, for FSMs with less than or equal to 512 states, the Q(RS) algorithm was, on average, 375 times faster than the GREEDY algorithm (with an average maximum of 400 fold and an average minimum of 350 fold). Additionally, the results show that as the number of inputs increases, both algorithms become slower, with the GREEDY algorithm slowing down faster. However, the Wilcoxon paired statistical analysis accepts the null hypothesis when $n = 32$. Considering Figure 11, we can deduce that the GREEDY and Q(RS) algorithms can generate RSs from small FSMs using a similar amount of time.

The results of the memory comparison (as shown in Figure 12) suggest that the Q(RS) algorithm can generate RSs while using significantly less memory than the GREEDY method. While this outcome is promising, it was also expected because the GREEDY algorithm necessitates a product automaton to process an FSM, which requires quadratic space.

### 5.4.2 Length of sequences

The results given in Figure 12 show that, on average, the Q(RS) algorithm generates RSs that are 3.75 times longer than those produced by the GREEDY method. We observed that with an increase in the number of inputs from two to four, the difference between the average lengths of RSs decreases from 5 fold to 2.5 fold. This decrease in the difference is due to the Q(RS) algorithm’s ability to explore more alternatives when searching for an RS, as there are more inputs. The Wilcoxon analyses (Table 5) and Cohen’s d test (Table 6) conducted on the lengths of sequences support the above findings by rejecting the null hypothesis.
The state-of-the-art approach (GREEDY) failed to process some test subjects. The Q-Graph framework increases the scalability by 100%. The proposed framework is 375 times faster, requiring 166 times less memory, but the generated RSs are 3.75 times longer on average than those produced by the GREEDY algorithm.

5.5 Results on RSs derived from partial FSMs

5.5.1 Time & memory

For partial FSMs, we used the PBF as the benchmark algorithm. The time comparison of these algorithms is given in Figure 14. The results suggest that regardless of the number of inputs, the two algorithms require a similar amount of time when the number of states is less than 256. This claim is also supported by the Wilcoxon test as shown in Table 5. Moreover, when the number of inputs is larger than two, and the number of states exceeds 4096, both algorithms fail to return RSs due to the memory limit (one GB). However, in other settings, the PBF algorithm computes RSs more quickly (12 times on average). Comparing the results given in [40], we can support this result due to two key points; first, as noted in [12], the PBF algorithm is designed for FSMs having a small number of inputs (less than 10) and its performance increases as the number of inputs decreases (in Section 5.6, we will show this phenomenon with an example). In contrast, because having more inputs increases the number of RSs of a given system, the Q(RS) algorithm can generate larger Q-graphs and increases its time requirement. The second reason is the differences in GPU technology in these experiments. The GPU used in experiments given in [40] has 48 CUDA cores, whereas, in this paper, we used a newer GPU having 1024 CUDA cores. The newer model features around 27% higher core clock speed, around 58% higher boost clock speed, and 21.3 times more pipelines, and the newer model has 4 times memory capacity. For detailed analysis, we refer the reader to NVIDIA’s specifications [80].

Figure 15 provides the memory usage trend for the algorithms. In the chart, we observe that the memory requirements of both algorithms increase with a similar trend. Because the y-axis of the chart is given in the log10 scale, both algorithms’ memory requirements are increasing exponentially with the number of states, where the Q(RS) memory requirement increases slightly faster. This is also confirmed by the Wilcoxon test, where the results are less than 0.05, suggesting a difference. Cohen’s d test also suggests a strong difference.

5.5.2 Length of sequences

Finally, the length of the constructed RSs is given in Figure 16. Results suggest that despite the Q(RS) algorithm exploiting a heuristic and PBF algorithm being a brute-force algorithm, both algorithms generate RSs with comparable lengths. We noted that 26% of the time, the Q(RS) algorithm generates longer RSs (33% longer on average). Finally, these findings are supported by the Wilcoxon hypothesis test and Cohen’s d test in Table 5, and Table 6, respectively.
5.6 Results for benchmark FSMs

5.6.1 Results on the Engine Status Manager

We provided \(v_1\) of ESM to the ST, Q(PDS), GREEDY and the Q(RS) algorithms and \(v_2\) of ESM to the PBF and Q(RS) algorithms. However, ST could not process the model and failed to return a PDS. Q(PDS) algorithm returned an empty PDS sequence declaring that the FSM does not possess a PDS; during this experiment, the Q(PDS) algorithm used 405 megabytes of RAM with 667 milliseconds. Similarly, the GREEDY and PBF algorithms could not generate RSs due to memory and time constraints. The Q(RS) algorithm generated RSs for both \(v_1\) and \(v_2\), with lengths of 352 and 375, respectively, resetting the system to state s524. For \(v_1\), the Q(RS) algorithm generated the sequence in 175 milliseconds using 367 MBs of RAM, and for \(v_2\), the Q(RS) algorithm generated the sequence in 189 milliseconds using 395 MBs of RAM.

5.6.2 Results on FSMs modelling circuits

We summarised the experiment results on the remaining benchmark FSMs in Tables 7 and 8. We noted that in all experiments, the time required to process FSMs are similar; therefore, we did not provide a time dimension. Results regarding PDS are given in Table 7. The \(y\) axis of the table gives the ratio of memory (and length) values obtained from Q(PDS) and ST. Value 1 in the chart means no difference. A value less than one indicates that the memory (or length) value is higher when ST is used, and a value larger than one indicates that the memory (or length) value is higher when Q(PDS) is used.

The chart reveals that 16 out of 59 FSMs possess PDS, and the ST algorithm failed to produce PDSs for five FSMs (33%) for which the PDS sequences exist (please see the FSMs where the length comparison is empty). Moreover, with respect to PDS generation, except for three FSMs, we observe that the Q(PDS) algorithm required a relatively higher memory than ST. Moreover, in 35% of the cases, the Q(PDS) algorithm generated longer PDSs.

The results related to RS can be found in Table 8, where the ratio of memory (and length) values obtained from the Q(RS) and the GREEDY algorithms are presented on the \(y\) axis of the table. A value of 1 in the chart indicates no difference, a value less than one suggests that the memory (or length) value is higher when GREEDY is used, and a value greater than one suggests that the memory (or length) value is higher when Q(RS) is used. The chart confirms
the results gathered from randomly generated FSMs. The Q(RS) algorithm, except for one FSM, required less memory space than the GREEDY algorithm. This is important as requiring less memory space makes the Q(RS) algorithm a better option for retrieving RSs from large FSM models that the GREEDY algorithm cannot process. However, the length of the RSs introduces a trade-off as the Q(RS) algorithm produces relatively longer RSs.

### Table 7

<table>
<thead>
<tr>
<th>FSM Name</th>
<th>Memory</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbtas</td>
<td>ex4</td>
<td></td>
</tr>
<tr>
<td>ex4_with_loops</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex4_with_sink</td>
<td></td>
<td></td>
</tr>
<tr>
<td>keyb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lion_with_loops</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mark1_with_loops</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mark1_with_sink</td>
<td></td>
<td></td>
</tr>
<tr>
<td>me</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opus_with_loops</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opus_with_sink</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pma_with_loops</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pma_with_sink</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shiftreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tax</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PDS results of benchmark experiments.

6 Extension to Timed FSMs

Timed FSMs (tFSMs) extend the FSM formalism with a single clock, which measures how long the tFSM has been in its current state. Such FSMs serve as the underlying model of many real application examples such as the TFTP [81], TCP [82], INRES [83], SCP [84], and they have recently been used to model the behaviour of common IoT edge devices [85]. At any point in the execution of a tFSM, the tFSM is in a state $s$ and has a value $t$ for the clock. We combine these pieces of information to define the notion of a location $\ell(s, t)$. We use the term location, as opposed to state, so that we can be clear whether we are referring to an FSM state (without a clock value) or a pair containing an FSM state and a clock value.

A state $s$ of a tFSM has a local time limit $T_s$; if the tFSM is in state $s$ and the clock value reaches $T_s$, then a transition occurs, moving the tFSM to a different state $s'$. There are, therefore, three types of transitions: Transitions that correspond to those found in FSMs, which are triggered by inputs; Transitions that involve the increase in the value of the clock by one unit of time and (timeout) transitions caused by the clock reaching the local time limit $T_s$ of the current state $s$. The clock is reset to zero whenever the tFSM takes a transition that is triggered by either an input or the local time limit being reached.

The behaviour of a tFSM will be defined in terms of transitions between locations. However, we will formalise tFSMs by essentially defining an FSM as before and adding the required additional information. There are several reasons for taking this approach, instead of directly defining transitions between locations. First, this is consistent with the work we build upon and also will allow us to reuse definitions and notation introduced earlier for FSMs. Second, if a developer is already familiar with FSMs, then, in order to use this formalisation, they simply need to add timeouts to states.

In the following, we first describe the tFSM model and formalise the behaviours of tFSMs. We then show that the Q-Graph framework can derive RSs and PDSs from tFSMs. Later in this section, we will provide the results of related experiments, providing empirical evidence that the proposed approach remains applicable in the timed settings.

#### 6.1 The timed FSM model

A timed FSMs (tFSMs) $tM = (S, X, Y, \delta, \lambda, s_0, \Delta)$ is an FSM with a single clock. It has a finite set of states $S$, inputs $X$ and outputs $Y$. The $\delta$ and $\lambda$ functions are identical to those defined for ordinary FSMs. Hence, the definitions given in Section 3 apply (e.g., defined input sequence, undefined input sequence, an extension of functions to a set of states, and sets of a set of states etc.). Whenever $tM$ reaches a state $s \in S$, the clock starts to measure the time spent at $s$. This introduces the term location $\ell$. A location $\ell(s, t_s)$ is a pair of state $s$ and local time $t_s$, where $t_s \in \mathbb{N}$ and $\mathbb{N}$ is the set of non-negative integers. When the tFSM reaches state $s$, the local time is set to 0; hence, the location is $\ell(s, 0)$. We summarised the terminology we introduced in this section in Table 9.

As long as $tM$ stays in $s$, $tM$ increments $t_s$ in system-specific time intervals, i.e., $\ell(s, 0), \ell(s, 1), \ell(s, 2), \ldots \ell(s, T_s - 1))$. Here $T_s$ is the local-time limit where $T_s \in (\mathbb{N} \cup \{\infty\})$. The difference between the local-time limit and the local time is the remaining time $r_s$, i.e., $r_s = T_s - t_s$. The timed transition function $\Delta : S \times (\mathbb{N} \cup \{\infty\}) \rightarrow S$. The timed transition function $\Delta : S \rightarrow S \times (\mathbb{N} \cup \{\infty\})$ maps a
state $s$ to the pair $(s', T_s)$ in which $T_s$ is the local time limit of $s$ and $tM$ moves to state $s'$ if the clock value reaches $T_s$ when $tM$ is in state $s$.

Figure 17 depicts a tFSM, taken from [85], modelling the behaviour of the passive infrared motion sensor (PIR), which measures the infrared light radiating from objects or human bodies nearby to detect whether the user is approaching or not [86]. The sensor is connected to a microcontroller (MC) that, in turn, uses Wi-Fi to connect to a home router which is connected to the Internet. Upon detecting motion, the MC uses the MQTT protocol to send a message to the application server and potentially to the apps running on a consumer’s laptop or a mobile phone. The MC must first connect to the WiFi network and then connect to an MQTT Broker using TCP. Once connected to WiFi and the MQTT broker, the MC continuously checks for a movement signal from the PIR motion detector. Once a signal is detected, debouncing is done by sleeping for a few milliseconds. For each movement event, an MQTT message is then published. Detailed input and output descriptions are given in [85]. The PIR tFSM is partial, and we completed it by introducing self-loop transitions having output $N$. For this example, the transition from $s_0$ leads to state $s_1$ with a local time limit 2, enabling the sequence that starts at location $\ell(s_1, 0)$, moves on (due to a timer step) from $\ell(s_0, 1)$, and after waiting one unit of time by changing the state from $s_0$ to state $s_1$. If $T_s = \infty$, then $s = s'$, meaning that $tM$ can stay in $s$ infinity long waiting for input.

Let $tM$ be at location $\ell(s, t_s)$, and $x$ be a defined input at $s$ where $\delta(s, x) = s'$, $\chi(s, x) = y$, and $\Delta(s) = (s'', T_s)$. Then if the input $x$ is applied when $t_s < T_s$, then the $tM$ changes its state to $s'$ producing the output $y$, and updates the location to $\ell(s', 0)$. However, if the local time $t_s$ becomes equal to $T_s$ before the input is applied, the machine moves to state $s''$, and the location is set to $\ell(s'', 0)$.

For example, assume the tFSM $tM$ given in Figure 17 is at state $s_{52}$, and the location is $\ell(s_{52}, 1)$. If input $x_{22}$ is applied before $tM$ moves to $\ell(s_{52}, 0)$ then $tM$ updates its state and location to $s_{41}$ and $\ell(s_{41}, 0)$ producing output $y_{12}$. However, if no input is applied, the state and location will be updated to $s_{41}$ and $\ell(s_{41}, 0)$ without producing any output.

### 6.2 Behaviour of timed FSMs

We can define the behaviour of a tFSM with respect to time by a next location function $\chi : S \times N \times N \rightarrow S \times N$. Given a location $\ell(s, t_s)$, the function $\chi$ specifies the next location that the tFSM will reach after waiting some time $t'$ (time input) from a location. In the following let $\Delta(s) = (s', T_s)$.

$$\chi(\ell(s, t_s), t') = \begin{cases} (\ell(s', 0), \text{if } t_s + t' = T_s) \\
(\ell(s, t_s + t'), \text{if } T_s = \infty \land t_s + t' < T_s), \\
(\varepsilon, \text{if } t_s + t' > T_s). \end{cases}$$

The last rule states that $\chi$ is defined when $t'$ is less than or equal to $t_s$, i.e., $t' \leq t_s$. If $t'$ is larger than $t_s$ then $t'$ is an undefined input and $\chi$ returns $\varepsilon$.

We can extend $\chi$ to a set of locations as follows. If $L = \{\ell(s, t_s), \ell(s', t_s'), \ldots\}$ is a set of locations then $\chi(L, t) = \{\chi(\ell(s, t_s), t) | \ell(s, t_s) \in L\}$. One important constraint on $t$ is that $t$ must be less than or equal to the minimum remaining time value $t_{min}(L)$ where $t_{min}(L) = \min_{\ell(s, t_s) \in L} t_s$. If $t$ exceeds $t_{min}(L)$ then $\chi$ will return $\varepsilon$. Furthermore, we can extend $\chi$ to a set of sets of locations. If $L = \{L_1, L_2, \ldots\}$ is a set of sets of locations then $\chi(L, t) = \bigcup_{L \in L} \chi(L, t)$. Again, $t$ must be less than or equal to the minimum remaining time value in $L$; otherwise, the function will return $\varepsilon$.

As with an FSM, the labels of a sequence of consecutive transitions (timed or untimed) defines a trace, and the input labels of the trace define a timed-input sequence: a sequence of inputs and time values, i.e., $\bar{x}_t \in (X \times N)^*$. For example, consider tFSM $tM$ in Figure 17. Assume we want to reach location $\ell(s_{52}, 0)$ from $\ell(s_{51}, 1)$. This can be done by waiting 1 unit of time, switching to state $s_{41}$, and then applying input $x_{61}$. This defines timed input sequence $\bar{x}_t = 1x_{61}$. When executing these transitions, $tM$ follows $\ell(s_{51}, 1)\ell(s_{41}, 0)\ell(s_{61}, 0)$ (please note $T_s(s_{52}) = 2$ and $\Delta(s_{52}) = (s_{42}, 2)$). However, if the input sequence is $\bar{x}_t = x_{22}x_{61}$, then $tM$ reaches $\ell(s_{51}, 0)$ from $\ell(s_{51}, 0)$ and produce the output $y_{12}y_{10}$. One key observation is that executing a timed transition does not cause a tFSM to produce an output.

If a timed input sequence contains undefined timed input or undefined input, then the timed input sequence is said to be an undefined timed input sequence. Otherwise, it is a defined timed input sequence. If a timed input sequence is undefined, then $\delta$, $\chi$, and $\Delta$ return $\varepsilon$.

To simulate the behaviour of a tFSM under timed input sequences, we use functions $U$ and $L : U : S \times N \times (N \cup X \cup \{\varepsilon\}) \rightarrow S \times N \cup \{\varepsilon\}$ combines the next state ($\delta$) and timed next state ($\chi$) functions and is defined by:

$$U(\ell(s, t_s), k) = \begin{cases} \ell(\delta(s, k), 0), \text{if } k \in X, \\
\chi(\ell(s, t_s), k), \text{if } k \in N \end{cases}$$
We can extend U to timed input sequences as follows. Let \( kk' \ldots \) be a timed input sequence, then \( U(\ell(s, t_a), kk' \ldots) = U(U(\ell(s, t_a), k), k', \ldots) \). Moreover, U can be extended to a set of locations (U) and a set of sets of locations (U) using extended variations of \( \delta \) and \( \chi \). In the above example, if we let \( x = x_1, 2 \) then we have \( U(\ell(s_1, 0), x_1, 2) = \ell(s_3, 0) \) because \( U(\ell(s_1, 0), x_1) = \ell(\delta(s_1, 1), x_1) = \ell(\ell(s_2, 0), 2) = \ell(s_3, 0) \).

We will use L to represent the output function for timed input. L is in the given form: \( L : S \times \mathbb{N} \times (\mathbb{N} \cup X \cup \{\varepsilon\}) \to Y \cup \{\varepsilon, \varepsilon\} \). The following defines function L.

\[
L(\ell(s, t_s), k) = \begin{cases} 
\lambda(s, k), & \text{iff } k \in X \cup \{\varepsilon\}, \\
\varepsilon, & \text{iff } k \in \mathbb{N}.
\end{cases}
\]

Extending L to timed input sequences is straightforward, i.e., \( L(\ell(s, t_s), kk' \ldots) = L(\ell(s, t_s), k) \cdot L(U(\ell(s, t_s), k), k', \ldots) \). Following the same example, if \( x = x_1, 2 \) and tM is at \( \ell(s_0, 0) \), then it returns \( L(\ell(s_0, 0), x_1) = L(\ell(s_0, 0), 2) \cdot L(\chi(\ell(s_0, 0), 2), x_1) \cdot L(x_1) = \lambda(s_1, x_1) = y_1 \). Note that we can extend L to a set of locations by using \( \lambda \) and \( \chi \).

In the following section, we define RS and PDS for tFSMs and show how the proposed algorithm can derive RSs and PDSs from tFSMs.

### 6.3 Constructing reset sequences from tFSMs.

We first define reset sequences for tFSMs. First, we let set \( \mathcal{L}^0 \) be the set of all possible locations that can occur in \( tM \).

**Definition 6.1.** Let \( tM = (S, X, Y, \lambda, \delta, s_0, \Delta) \) be an tFSM. A timed input sequence \( x_1 \) is a timed reset sequence (tRS) for \( tM \) iff \( U(L, tM, x_1) = 1 \).

We are to construct a timed input sequence that brings the underlying tFSM to a common location, regardless of the initial location.

For example, the timed input sequence \( x_1 = x_1, 2 \) is a tRS for the tFSM given in Figure 17, \( U(\ell(s_0, 0), x_1, 2) = \ell(s_4, 0) \). We introduce an action \( a_{x_1} \) for \( x_1 \in X \) and \( a_2 \) to \( A \) for \( 2 \leq r_s \).

By abusing the notations \( (i), st(\{) \) introduced in Section 4.2, we use one-to-one and onto functions to denote corresponding inputs and set of states: \( i(\{) \) maps an input from set \( X \cup \{2, \ldots, r_s\} \) to the corresponding action \( a \) of the MDP P and vice versa, and we let \( st(\{) \) map a set of locations \( C' \in \mathcal{P}(S) \times r_s \) of the tFSM \( tM \) to the corresponding state \( K' \in \mathcal{Z} \) of the MDP P and vice versa. A state change from \( K \) to \( K' \) is triggered by an application of an action \( a \). We define the probability function \( P_t(K'|K, a) \) as follows, let \( U(st(K), i(a)) = st(K') \).

In this setting, the agent’s aim is to find a sequence of actions (consisting of inputs and timed inputs) that takes the agent from an MDP state \( K \in \mathcal{Z} \) to another MDP state that is associated with \( L \) such that \( |L| = 1 \). If input given by \( i(a) \) is not defined for the set of tFSMs given by \( st(K) \), then the reward function returns NaN. For the defined actions, the reward function to compute a tRS is similar to the return of RS given in Section 4.2.

\[
P_t(K'|K, a) = \begin{cases} 
1, & \text{iff } U(st(K), i(a)) = st(K'), \\
0, & \text{otherwise}.
\end{cases}
\]

In implementing the tRS problem using Q-Graph, the initial node of the Q-Graph has \( st(\mathcal{L}^0) \) as its MDP state.

### 6.4 Constructing preset distinguishing sequences from tFSMs.

**Definition 6.2.** Let \( tM = (S, X, Y, \lambda, \delta, s_0, \Delta) \) be a tFSM. A timed input sequence \( x_1 \) is a timed preset distinguishing
sequence (IPDS) for $tM$ if for every pair of $s$ and $s'$ of distinct states of $tM$ and every pair of locations $(\ell, \ell')$ where $\ell \in L_s$ and $\ell' \in L_{s'}$, we have, $L(\ell, \bar{x}_t) \neq L(\ell', \bar{x}_t)$.

For example, the timed input sequence $\bar{x}_t = x_0 x_2 x_3 x_5 x_1 x_0 x_2 x_{10}$ is a tPDS for the tFSM given in Figure 17. Let $l_{s_0}, l_{s_1},$ and $l_{s_2}$ be local times for states $s_0, s_1,$ and $s_2$, then $L(l_{s_0}, l_{s_0}), \bar{x}_t) = NaN NaN y_1 N y_N$, $L(l_{s_1}, l_{s_1}), \bar{x}) = y_1 N NaN y_3 N,$ and for example $L(l_{s_2}, l_{s_2}), \bar{x}) = NaN$ $N y_N y_3 N$.

We represent the tPDS construction problem as an MDP $P = (Z, A, \mathcal{P}_t, R_{IPDS})$ as follows. We represent a set of sets of locations $\hat{L}$ of the tFSM as a state $K \in Z$ of $P$. Next, for each input $x$ of tFSM and each unique value in $1, 2, \ldots, r_{\mathcal{F}}$, we introduced an action and form set $A$.

Again by abusing notations introduced in Sections 4.2, and 6.3, we use $i(.)$ to convert input from set $X \cup \{1, 2, \ldots, r_{\mathcal{F}}\}$ to an action $a \in A$ and vice versa in the usual way. Moreover, let $st(.)$ be one-to-one and onto function that converts a set of sets of locations $\hat{L}$ of tFSM to a state $K$ of $P$, and vice-versa.

The state transitions of $P$ are given as follows.

$$\mathcal{P}_t(K'|K, a) = \begin{cases} 1, & \text{iff } s(t(K), i(a)) = s(K') , \\ 0, & \text{otherwise} . \end{cases}$$

The reward function for finding tPDSs is the same as the function introduced for PDSs in Section 4.3. The only difference is that if the time input given by action $a$ is not defined for the underlying set of locations, it returns $NaN$. Otherwise, it returns $-1$ because the underlying tFSM would not produce an output if a timed input is applied and, by doing so, cannot contribute to distinguishing states. We illustrate this in Figure 20 for the tFSM given in Figure 17. Note that the action corresponding to timed input 1 causes locations to merge at $\{s_4, 0\}$, so the reward is $NaN$. An input that does not cause states to distinguish (input $x_2$) receives $-1$. For consistency, we will use $R_{IPDS}$ to refer to the reward function when we use it for tFSMs. As in the case of the tRS problem, in implementing the tPDS problem using Q-Graph, the initial node has $s(t(L^0))$.

6.5 Experimental settings for tFSMs

We conducted experiments on randomly generated partial tFSMs to analyse algorithm performance for tPDS and tRS generation. As in the case of FSMs, the evaluations were focused on examining the execution time, memory requirements, and sequence length.

The tFSMs were created using Algorithm 5 with the following steps: given values of $n$, $i + 1$, and $p = 10$, a partial FSM was generated with $n$ states, $i + 1$ inputs and $i$ outputs. After this step finished, we defined the transitions in the following way. We selected the $i + 1$th input as the timed input. Then, for every state, we selected the transition with the input label $i + 1$. If this transition ended in the same state, we set the label to $\infty$. However, if it ended in a different state, the label was set to a randomly generated value between 1 and 100. That is, the local-time limit values of the states were between 1 and 100.

The Q(PDS) and Q(RS) algorithms were then applied to these tFSMs, and the tFSMs were retained if they possessed a tRS and/or a tPDS. For studying tPDSs, for each $n$, $i$ values from $\{32, 64, 128, 256\}$ and $\{3, 4, 5\}$ we created one hundred tFSMs. To study tRSs, we could only create one hundred tFSMs for each $n$, $i$ value from $\{32, 64\}$ and $\{3, 4, 5\}$.

Therefore, during the experiments, we used 1800 tFSMs in total.

To run the tFSMs on the benchmark algorithms, we unfolded them to abstract their time behaviour, as described in [36], before testing. This process of unfolding the tFSMs is time-consuming, and the resulting unfolded tFSMs (u-tFSMs) have an increased number of states proportional to the maximum timeout value present in the underlying tFSMs. We modified the GREEDY and ST algorithms so that they can receive a time value as input and generate sequences for a specific subset of the states of the u-tFSMs.

Note that the step in which we unfold the tFSMs is only required for the benchmark algorithms, as the Q-Graph algorithm processes tFSMs. However, during the experiments, we excluded the time required to unfold FSMs as we only wanted to compare the time needed to derive PDSs and RSs. Size comparisons of the u-tFSMs obtained from tFSMs with varying numbers of states, denoted as $n$, are presented in Figures 21 and 22. The average number of states of the u-tFSMs is shown on the y-axis, with $n$ indicated on the x-axis.
6.6 Results on tPDSs derived from partial tPDSs

The findings related to tPDSs are presented in Figures 23 (time comparison), 24 (memory comparison), and 25 (length comparison). Our method stands out due to its unique capability of computing a tPDS or tRS from tFSMs. Our analysis revealed that the ST algorithm generated tPDSs from u-tFSMs when the state count was 32, and the input count was four or five. In other words, the Q(tPDS) increased the scalability by 4 times. However, the ST algorithm faced memory limitations (1GB) and failed to produce results for other cases. Conversely, the Q(tPDS) algorithm could handle tFSMs from all test subjects.

Regarding performance, Q(tPDS) exhibited an average speed improvement of 925 fold and a memory reduction of 100 fold compared to the ST algorithm (the time spent on unfolding is excluded). However, the tPDSs generated by the ST algorithm (excluding sequences of length zero) were approximately 2.5 times shorter than those produced by Q(tPDS).

6.7 Results on tRSs derived from partial tPDSs

The findings related to tRSs are presented and summarised in Figures 26 (time comparison), 27 (memory comparison), and 28 (length comparison). The results are promising. The PBF algorithm could generate RSs from u-tFSMs when the number of states was 32, and the number of inputs was three. The Q(tRS), on the other hand, could generate tRSs from all test subjects, improving the scalability by 6 times. For the test subjects having 32 states and three inputs, the Q(tRS) algorithm was 700 times faster than the PBF algorithm and used 3 times less memory (the time spent on unfolding is excluded). As the PBF algorithm deploys a brute-force search, the PBF algorithm generated shorter (20% on average) tRS.

7 DISCUSSION

7.1 Answers to research questions

7.1.1 Answer for RQ-1.a: What sort of reward functions can be used to derive resetting and state-verification sequences from different types of FSMs?

We introduced a Q learning algorithm for MBT, which is modular and searches for test sequences based on the reward function. For deriving RSs and PDSs from different kinds of models, we introduced four reward functions, each focusing on different kinds of properties. We provided these functions in Sections 4.3, 4.2, 6.3, and 6.4.

7.1.2 Answer for RQ-1.b: The framework will be based on Q-graph approach, and the Q-Graph approach is stochastic, so will the proposed framework behave differently in different runs when it is exposed to the same inputs? That is, is the new approach functionally stable?

We followed the standard procedure [39]: we randomly picked an FSM having 8192 states and five inputs and outputs and repeated the PDS and RS generation tasks at most 100 times where each could take at most 100 episodes where an episode ends either when the target sequence is found, or the agent picked an action that corresponds to an undefined input symbol. For each, we measured the Cumulative Rewards produced by the underlying reward function. Results are given in Figure 29 where the y axis gives the normalised cumulative reward. We observe that, generally, the Q-Graph algorithm can quickly gather positive rewards after a couple of episodes with increasing consistency. Moreover, when the framework computes RSs, the rewards converge to one quicker than when the Q-learner computes PDSs. However, in both studies, we observe that the algorithm converges to the optimum reward as the number of episodes increases. These observations are confirmed by the confidence interval (bold line in Figure 29) with a confidence interval of 0.95. Another important observation is that the proposed Q-Graph framework avoids constructing the Q-table, which is the fundamental data structure of tabular Q-learning framework. In comparison, the Q-table framework would need $2^n \times i$ number of cells to derive test sequences from FSMs with $n$ states and $i$ inputs. Instead, when $n = 16384$ and $i = 5$, the framework only requires 100 MBs of RAM, a significant reduction (more than 99.99%) in memory requirement compared to storing a table having $2^{16384} \times 5$ cells.

7.1.3 Answer for RQ-2.a: How efficient is the proposed approach regarding memory and time requirement when compared to the state-of-the-art algorithms?

Experiments conducted on our FSM subjects are conclusive and indicate that our proposed method is superior to the existing approaches in many ways. First, we have shown that the proposed algorithm can generate RSs or PDSs from various kinds of FSMs that no other method could process. Second, the proposed method can generate PDSs using less memory faster. Regarding RSs, we observe that the PBF algorithm is generally faster with better memory utilisation than our approach. However, the PBF algorithm could not process a real FSM model where the proposed method could generate an RS.

7.1.4 Answer for RQ-2.b: How effective is the proposed approach, in terms of the sizes of the resetting and state-verification sequences, compared to the sequences generated by other existing algorithms?

Experiment results are very promising. The proposed algorithm generates PDSs and RSs from partial FSMs with similar lengths. This is surprising because all the existing algorithms are brute-force, and they compute the shortest possible sequences. Moreover, the proposed algorithm can also produce PDSs from complete FSMs with similar lengths compared to the existing brute-force method. We provided the results of these experiments in Sections 5 and 6.

7.2 Threats to validity

We identified a number of threats to the validity of our results. The first potential threat is selection bias, which refers to how well the test subjects represent the overall population. We utilised computer-generated FSMs and FSMs obtained from industrial case studies to mitigate

7. Considering the results given in [40], this is thanks to the GPU technology used in this work.
Fig. 23. The average time required to construct tPDSs from partial randomly generated tFSMs.

Fig. 24. The average memory required to construct tPDSs from partial randomly generated tFSMs.

Fig. 25. The average lengths of tPDSs constructed from partial randomly generated tFSMs.

Fig. 26. The average time required to construct tRSs from partial randomly generated tFSMs.

Fig. 27. The average memory required to construct tRSs from partial randomly generated tFSMs.

Fig. 28. The average lengths of tRSs constructed from partial randomly generated tFSMs.

Fig. 29. Normalised cumulative rewards collected by Q-Graph framework while constructing RSs and PDSs from a given FSM.

8 Conclusion

The ever-increasing complexity and diversity of software systems pose formidable challenges in testing. The intricacies involved in computing reset and state identification sequences create significant hurdles. However, recent advancements in artificial intelligence (AI) technology have provided a myriad of domain-agnostic solutions. This study presents an AI framework, termed the Q-Graph framework, designed explicitly for Model-Based Testing (MBT). Our framework demonstrates modularity, enabling the construction of reset and state identification sequences from various finite state machines (FSMs) by leveraging distinct reward functions. Through comprehensive experiments conducted on randomly generated and real-life FSMs and rigorous comparative analysis, we establish the unparalleled efficacy of our proposed framework.

Nonetheless, numerous avenues for future research remain unexplored. Firstly, addressing the computational overhead associated with deriving such sequences from non-deterministic systems is imperative, warranting an extension of our proposed framework to encompass systems featuring non-deterministic components. Secondly, as explained in Section 1, diverse test sequence types call for expanding our methodology to incorporate such sequences, representing a promising avenue for further investigation. Finally, it is noteworthy to highlight the significant value we have observed in accelerating the current approach by leveraging general-purpose graphics processing units (GPUs).

Acknowledgements We have been supported by the UKRI Trustworthy Autonomous Systems Node in Verifiability, Grant Award Reference EP/V026801/2 and EPSRC:
RoboTest: Systematic Model-Based Testing and Simulation of Mobile Autonomous Robots, EP/R025134/1. Khaled El-Fakhî has been partially supported by AUS FRG23-R-E39 Grant. Mohammad Reza Mousavi has been partially supported by the EPSRC project on Verified Simulation for Large Quantum Systems (VSL-Q), grant reference EP/Y005244/1 and the EPSRC project on Robust and Reliable Quantum Computing (RoRaQ), Investigation 009 Model-based monitoring and calibration of quantum computations (ModeMCG), grant reference EP/W032635/1, as well as the King’s Quantum grant provided by King’s College London.

REFERENCES


