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 $\mathcal{Q}$-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

Automated 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 Océ 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

- *U.C. Türker was with the School of Computing and Communications, Lancaster University, Lancaster, UK, LA1 4YW. E-mail: u.turker@lancaster.ac.uk*
- *Robert M Hierons was with the Department of Computer Science, The University of Sheffield, Sheffield, UK.*
- *Khaled El-Fakih was with the Department of Computer Science and Engineering, American University of Sharjah, UAE.*
- *Mohammad Reza Mousavi was with the Department of Informatics, Kings College London, UK.*
- *Ivan Y. Tyukin was with the Department of Mathematics, Kings College London, UK.*

Manuscript received July 19, 2023; revised XXX YYYY.

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 $\mathcal{Q}$-learning, we estimate the expected cumulative reward of taking a particular action in a particular state. Because of this nature, $\mathcal{Q}$-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 $\mathcal{Q}$-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:

Seq. Type	FSM type	Approach in [40]	Proposed approach
RS	Complete FSMs	✔	✔
	Partial FSMs	✔	✔
	Timed FSMs	✗	✔
PDS	Complete FSMs	✗	✔
	Partial FSMs	✗	✔
	Timed FSMs	✗	✔

TABLE 1
Capability comparison for the proposed framework

RQ-1 : *Is it possible to define a general $\mathcal{Q}$-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 $\mathcal{Q}$-graph approach, and the $\mathcal{Q}$-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 $\mathcal{Q}$-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;
2) In the previous work, we proposed a stand-alone application with no stability analysis. In this work, we introduce a general RL framework for MBT and analyse its stability;
3) In contrast to our previous work, we allow models to have outputs as well as inputs;
4) We design and conduct experiments to empirically evaluate our algorithms and answer our research questions.

Note that in our previous work, we investigated the problem of devising SSs from a finite automaton that essentially is an FSM that does not include outputs. SSs

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 $\mathcal{Q}$-Graph framework. We describe the experiments and results for un-timed FSMs in Section 5. Afterwards, we demonstrate how the $\mathcal{Q}$-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 RSS 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 Güniçen 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][1], 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

1. Except for one class of FSMs [54].

Symbols	Definitions
$\hat{S}$,s_0	Set of sets of states, the initial state
S',S	Set of states, set of states
X,Y,x,y,$\bar{x}$,$\bar{y}$	Set of inputs, set of outputs, input, output, input sequence, output sequence.
ϵ, ε	Empty input (or output) (sequence), undefined input (or output) (sequence).
$\hat{\delta}$, δ, λ	Transition function for a set of sets of states, transition function, output function.
$pow(.)$	Power set of (.).

TABLE 2
List of symbols regarding FSMs

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], [64]. 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 ($s_0 \in S$), finite set of inputs (X) and a finite set of outputs (Y). When an input $x \in X$ is applied, when the FSM is in state s, the FSM produces an output $y \in Y$ and changes its current state to some $s' \in S$. We summarised the terminology we used for FSMs in Table 2.

In Figure 1, we give a complete FSM M_1 that models a `ShiftRegister` circuit [41]. With respect to FSM M_1, if a tester applies input x_1 when the FSM is in state s_0, then the FSM will produce output y_0 and change its state to s_1. 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 $start(\tau) \in S$, an input symbol $in(\tau) \in X$, an ending state $end(\tau) = \delta(s, x)$, and an output symbol $out(\tau) = \lambda(s, x)$. Thus, we can represent an FSM using the tuple $M = (S, X, Y, \delta, \lambda, s_0)$. If δ and λ are defined on all pairs in $S \times X$ then M is a *complete FSM*; otherwise M is a *partial FSM*. If M is complete then δ has type $S \times X \rightarrow S$ and λ has type $S \times X \rightarrow Y$. That is, δ is a function that maps (state $\times$ input) pairs to states, and λ is a function that maps (state $\times$ 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 $S \times X$ on which they are not defined. Thus, for partial FSMs we have functions types $\delta : S \times X \rightarrow S \cup \{\varepsilon\}$ and $\lambda : S \times X \rightarrow Y \cup \{\varepsilon\}$; if x is an undefined input for s then we have $\delta(s, x) = \varepsilon$ and $\lambda(s, x) = \varepsilon$. Similarly, if $x = \epsilon$, then we have $\delta(s, \epsilon) = s$ and $\lambda(s, \epsilon) = \epsilon$.

An input sequence $\bar{x} \in X^*$ is a sequence of inputs, i.e., $\bar{x} = x_1 x_2 \dots x_k$. If input sequence $\bar{x}$ is applied to a *complete* FSM in state s, the FSM performs a *walk* defined by a sequence of transitions $\tau_0 \tau_1 \tau_2 \dots \tau_k$. These are consecutive transitions: for all $1 \leq i < k$, we have that $end(\tau_i) = start(\tau_{i+1})$. If one concatenates the input symbols of these transitions, then one obtains the *trace* $x_1 x_2 \dots x_k$.

Suppose that $\bar{x}$ is a finite input sequence from $X^\star$; if the FSM M is complete, then for every state s of M, there is a walk starting from s that has trace $\bar{x}$. 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 $\bar{x}$ is a *defined input sequence* for s. An input sequence $\bar{x}$ is *defined* in state s, if for all non-empty prefixes $\bar{x}'x$ of $\bar{x}$, x is defined in the state reached from s with $\bar{x}'$.

We extend δ and λ to input sequences in the following way: let $\bar{x} = x\bar{x}'$ be an input sequence and $s \in S$ be a state of FSM M, then $\delta(s, x\bar{x}') = \delta(\delta(s, x), \bar{x}')$ and $\lambda(s, \bar{x}') = \lambda(s, x).\lambda(\delta(s, x), \bar{x}')$. Note that for a state s and input sequence $\bar{x}$, the functions returns symbol ε if $\bar{x}$ is not defined for s, and returns s if $\bar{x} = \epsilon$. Furthermore, we extend δ to a set of states $S' \subseteq S$ as follows: given an input x, output y and state set $S' \subseteq S$ then $\delta_y(S', x)$ 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 $\delta_y(S', x)$ to be ε if there is some state $s \in S'$ such that x is undefined in s and otherwise[2] $\delta_y(S', x) = \{s' | \exists s \in S'.\delta(s, x) = s' \wedge \lambda(s, x) = y\}$.

Let $\hat{S} = \{S', S'', \dots\}$ denote a set of sets of states, i.e., for all S' in $\hat{S}$ we have that $S' \subseteq S$. We use $\hat{\delta}(\hat{S}, x)$ to denote the outcome of applying an input to a set of sets of states as defined below:

$$\hat{\delta}(\hat{S}, x) = \begin{cases} \varepsilon, \text{ if } \exists S' \in \hat{S} \text{ s.t. } x \text{ is undefined for } S' \\ \emptyset, \text{ if } \exists S' \in \hat{S} \text{ s.t. } x \text{ is merging for } S' \\ \{\delta_y(S', x) | \exists s \in S'.y = \lambda(s, x), S' \in \hat{S}\}, \\ \text{otherwise.} \end{cases} ,$$

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 $\delta(s, x) =$

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.

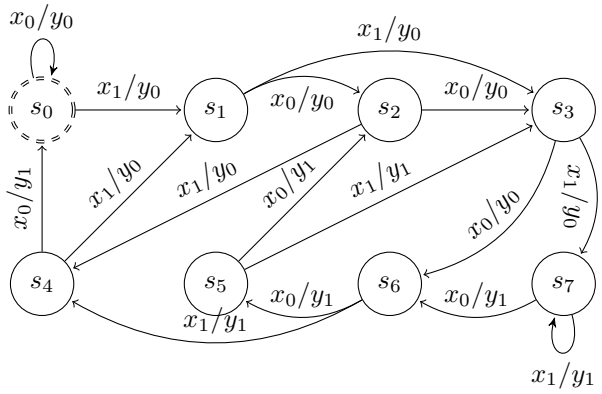

Fig. 1. Example FSM M_1 `ShiftRegister` from [41].

$\delta(s'x)$ and $\lambda(s,x) = \lambda(s'x)$. If x is merging for S' then the states in S' cannot be separated/distinguished by any input sequence that starts with x. The third case defines a set of sets of reached states according to the outputs that the FSM produces: for each possible output y, the set contains the set of states that can be reached from S' through input x and output y.

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 $\bar{x} = x_1x_1x_1x_1$, FSM 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 $M = (S, X, Y, \delta, \lambda, s_0)$ iff $\bar{x}$ 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 $\bar{x} = x_0x_1x_1x_1$ 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 $y_0y_0y_0y_0$, but it will produce $y_1y_0y_0y_0$ if M_1 was in state s_4. No pair of states will produce the same output sequence, and we can distinguish the states by using $\bar{x}$.

3.4 $\mathcal{Q}$-Learning environment

$\mathcal{Q}$-learning algorithms operate on a Markov Decision Process (MDP) [39]. An MDP is defined as a tuple $P = (\mathcal{Z}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where $\mathcal{Z}$ is a set of states, $\mathcal{A}$ is a set of actions, $\mathcal{P}$ is the probability function in the form of $\mathcal{P}(\mathcal{K}'|\mathcal{K}, a)$, that is for each action $a \in \mathcal{A}$ and state $\mathcal{K} \in \mathcal{Z}$, it defines the probability of moving to state $\mathcal{K}' \in \mathcal{Z}$ if action a occurs in state $\mathcal{K}'$. $\mathcal{R}$ is the immediate reward function in the form of $\mathcal{R} : \mathcal{Z} \times \mathcal{A} \times \mathcal{Z} \to \mathbb{R}$, 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.

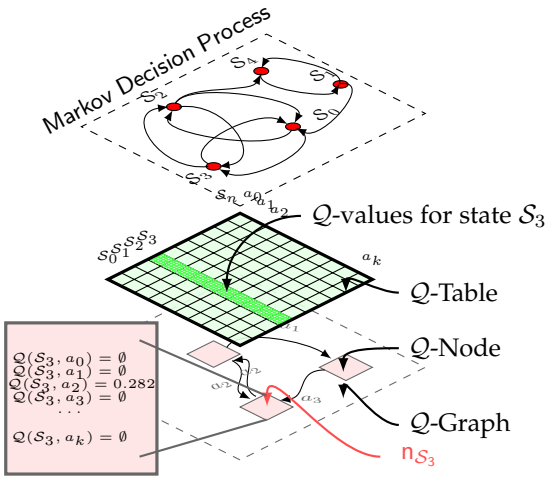

Fig. 2. From $\mathcal{Q}$-tables to $\mathcal{Q}$-Graphs.

$\mathcal{Q}$-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 $P = (\mathcal{Z}, \mathcal{A}, \mathcal{P}, \mathcal{R})$. Instead of estimating these unknown probabilities, the method uses a *value function* $\mathcal{Q}(\mathcal{K}, a)$ [68]. Let $\mathcal{K}'$ be a state reached from $\mathcal{K}$ using action a. The value function $\mathcal{Q}(\mathcal{K}, a)$ is recursively defined as:

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

where $\mathcal{K} \in \mathcal{Z}$, $a \in \mathcal{A}$, and $\mathcal{Q}(\mathcal{K}, a)$ is the value of applying action a at state $\mathcal{K}$, a' and $\mathcal{K}'$ are the next action and next states, respectively, $R(\mathcal{K}, a)$ is the immediate reward received after applying action a at state $\mathcal{K}$, 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 [68] 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 [39].

4 $\mathcal{Q}$-GRAPHS

A $\mathcal{Q}$-learning algorithm requires a function that calculates the quality of a state–action combination. 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 [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[3]. 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 $\mathcal{Q}$-Graph framework can reduce the memory requirement by more than 99.99% and can find optimum results.

4.1 The generic $\mathcal{Q}$-Graph algorithm

In practice, in tabular $\mathcal{Q}$-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 $\mathcal{K}'$ after applying action a from state $\mathcal{K}$ 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 $\mathcal{Q}$-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 $\mathcal{K}$ 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 $\mathcal{Q}$ learning. We illustrate this new framework in Figure 2 and formally define $\mathcal{Q}$-Graphs as follows.

Definition 4.1. A $\mathcal{Q}$-graph $\mathcal{Q} = (N, E)$ has a finite set of nodes ($\mathcal{Q}$-nodes) N and edges E. Each node $n \in N$ is associated with a state of the environment $\mathcal{K}$ denoted as $n_{\mathcal{K}}$ and a $\mathcal{Q}$-value ($\mathcal{Q}(\mathcal{K}, a)$) 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 $n_{\mathcal{K}'}$ from $n_{\mathcal{K}}$ using an action a. We summarised the new terminology we introduced for $\mathcal{Q}$-learning in Table 3.

A generic algorithm for the $\mathcal{Q}$-Graph framework is given in Algorithm 1. Note that the algorithm is similar to the generic tabular $\mathcal{Q}$-learning algorithm [39]. The first step of this algorithm is to select random $\mathcal{Q}$ values for each action of the initial node and create the initial node (Lines 1-4 of Algorithm 1). After initialising the $\mathcal{Q}$-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 ($n_{\mathcal{K}}$) 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 ϵ-greedy[4], collecting the immediate reward from the environment, and updating the $\mathcal{Q}$ 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 $\mathcal{Q}$-graph will keep information for all state-action pairs and it is guaranteed that the $\mathcal{Q}$-Graph converges to an optimum policy if sufficiently many nodes are explored. In the experiments, we will show that the $\mathcal{Q}$-Graph can reduce the space requirements by more than 99.9% and computes an optimum policy.

Symbols	Definitions
$\mathcal{Z}$, $\mathcal{A}$, $\mathcal{K}$	Set of MDP states, Set of actions, MDP state.
$\mathcal{P}$, $\mathcal{R}$	Transition probability function, Immediate Reward Function.
$\mathcal{Q}(\mathcal{K}, a)$	The quality value of taking action a at MDP state $\mathcal{K}$.
$\mathcal{Q}$, N, E, n, $n_{\mathcal{K}}$	$\mathcal{Q}$ graph, set of $\mathcal{Q}$ nodes, set of edges, a $\mathcal{Q}$ node, MDP state associated to $\mathcal{Q}$ node.
$st(.)$, $i(.)$	(MDP_state-set_of_FSM_states) conversion function, (MDP_action-FSM_input) conversion function.

TABLE 3
List of symbols regarding $\mathcal{Q}$-learning.

Algorithm 1: A generic $\mathcal{Q}$-Graph algorithm.

```
1  Initialise a Q−node n_0 with the initial state K_0.
2  foreach a ∈ A do
3      Assign a random value to Q(K_0, a) of n_0.
4  N ← N ∪ {n_0}.
5  while There is an episode do
6      n' ← n_0.
7      while Current episode do
8          foreach a ∈ 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
                       with a and P(n''_{K''}|n'_K, a).
12         Take an action a ∈ A using policy derived from Q
               (e.g. ε-greedy).
13         Record reward and next state K''.
14         Update n', and Q-values according to Q function.
15         if n'' is not terminal then
16             n' ← n'' where n'' is associated with K''.
17         else
18             Kill current episode.
```

4.2 Constructing resetting sequences using $\mathcal{Q}$-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

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

4. In ϵ-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 $\mathcal{Q}$ function.

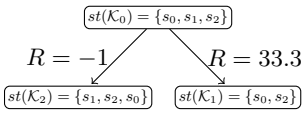

Fig. 3. Reward function for RS. The action leads to merging states, causing a positive reward.

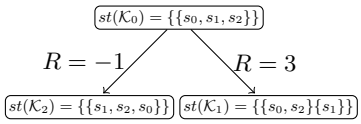

Fig. 4. Reward function for PDS. The action promotes distinguishing states, causing a positive reward.

to a state ($\mathcal{K}' \in \mathcal{Z}$) of the MDP P and each input $x \in X$ of FSM M 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 FSM M to the corresponding action $a \in \mathcal{A}$ of the MDP P and vice versa, and (ii) $st()$ maps a set of states $S' \in pow(S)$ of the FSM M to the corresponding state $\mathcal{K}' \in \mathcal{Z}$ of the MDP P and vice versa. In this setting, the $\mathcal{Q}$-learning agent's task is to reach a state $\mathcal{K}$ such that $|st(\mathcal{K})| = 1$. 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 an MDP state $\mathcal{K}$ due to two actions. One of them does not reduce the number of states $\mathcal{K}_2$, 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 $\mathcal{K}$ and a, we let

$$\mathcal{P}_{RS}(\mathcal{K}'|\mathcal{K}, a) = \begin{cases} 1, \text{ iff } \delta(st(\mathcal{K}), i(a)) = st(\mathcal{K}'), \\ 0, \text{ otherwise.} \end{cases}$$

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

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

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

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 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 $O(n^2)$ [42], and for a partial system, the bound is $O(n^2 * 4^{n/3})$ [72].

4.3 Constructing preset distinguishing sequences using $\mathcal{Q}$-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 $\hat{\delta}(\hat{S}, x)$ and $\hat{\delta}(\hat{S}, x')$ from $\hat{S}$ and selects one input (say x'). After this, it appends x' to the PDS under construction ($\bar{x}$) and forms a set $\hat{S}'' = \hat{\delta}(\hat{S}, x')$ 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 $\bar{x} = x, x', x'' \ldots$ is found, then $\hat{\delta}(\hat{\delta}(\hat{S}, x), x'x'' \ldots) = \hat{S}'$.

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 $a \in \mathcal{A}$ of P. However, a state $\mathcal{K} \in \mathcal{Z}$ of the MDP P is associated with a set of sets of states ($\hat{S}$) that is a subset of $power(S)$. Since FSM M has n states, the number of elements of an $\hat{S}$ associated with a $\mathcal{K} \in \mathcal{Z}$ 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 $\mathcal{Q}$-learning agent's task is to find a sequence of actions (inputs) that reaches an MDP state $\mathcal{K}'$ such that $|\hat{st}(\mathcal{K}')| = n$ from an MDP state $\mathcal{K}$ such that $|\hat{st}(\mathcal{K})| = 1$ by using the δ function of the underlying FSM. Here, $\hat{st}()$ is a one-to-one and onto function, which receives an MDP state $\mathcal{K}$ and returns a set of sets of states $\hat{S}$ and vice versa. According to this, using $i()$ and $\hat{st}()$ we define $\mathcal{P}$ of P as follows; for each $\mathcal{K}$ and a, we let

$$\mathcal{P}_{PDS}(\mathcal{K}'|\mathcal{K}, a) = \begin{cases} 1, \text{ iff } \hat{\delta}(\hat{st}(\mathcal{K}), i(a)) = \hat{st}(\mathcal{K}'), \\ 0, \text{ otherwise.} \end{cases}$$

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

The immediate reward function ($R_{PDS}(\mathcal{K}, a)$) penalises undefined and merging inputs and inputs that do not increase the number of sets in $\hat{st}(\mathcal{K})$. However, it rewards inputs that increase the number of sets in $\hat{st}(\mathcal{K})$ evenly. To achieve this, we consider the weight of input $x \in X$ ($\kappa(x)$) by using the following function. Let $\hat{S}'$ be a set of sets of states having more than one element due to the application of input x, then $\kappa(x) = \sum_{S', S'' \in \hat{S}'} ||S'| - |S''||$. 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 $\binom{n}{\eta}$ where $\eta = |S'|$ and $n = |S|$. So the immediate reward for the PDS problem is computed as follows in which $\mathcal{K}'$ is the next MDP state such that $\delta(\hat{st}(\mathcal{K}), i(a)) = \hat{st}(\mathcal{K}')$.

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

In Figure 4, we illustrate the reward function in action. We see that the action that partitions the set of states in MDP state $\mathcal{K}$ gets reward value 3, while the other action gets -1.

4.4 Accuracy and space requirement for $\mathcal{Q}$-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 $\mathcal{Q}$-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 $\mathcal{Q}$-Graph can find optimum policies for RS and PDS problems.

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

Corollary 5. Let M be an FSM having n states. Then the $\mathcal{Q}$-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** *l 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 $\delta(s, x) = s'$ and $\lambda(s, x) = y$.
5 **if** $p > 0$ **then**
6 Randomly select $(n * x)/100$ 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 untimed. 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.

Names	States	inputs
bbara	7	6
bbtas	6	4
beecount_with_loops	7	5
beecount_with_sink	8	5
cse_with_hidden_states_with_sink	34	19
cse_with_loops	16	19
cse_with_loops_with_hidden_states	41	19
cse_with_sink	17	19
dk14	7	8
dk15	4	8
dk16	27	4
dk17	8	4
dk27	7	2
dk512	14	2
ex2_with_hidden_states_with_sink	16	4
ex2_with_loops	10	4
ex2_with_sink	10	4
ex3_with_hidden_states_with_sink	25	4
ex3_with_loops	10	4
ex3_with_sink	10	4
ex4	14	24
ex5_with_hidden_states_with_sink	17	4
ex5_with_loops	9	4
ex5_with_sink	9	4
ex6_with_loops	8	20
ex6_with_sink	9	20
ex7_with_hidden_states_with_sink	8	4
ex7_with_loops	6	4
ex7_with_sink	6	4
keyb	19	24
keyb_with_hidden_states_with_sink	41	24
keyb_with_loops_with_hidden_states	41	24
lion9_with_loops	9	4
lion9_with_sink	10	4
lion_with_hidden_states_with_sink	5	4
lion_with_loops	4	4
lion_with_loops_with_hidden_states	4	4
lion_with_sink	5	4
mark1_with_loops	13	15
mark1_with_sink	13	14
mc	4	8
opus_with_loops	9	11
opus_with_sink	10	11
pma_with_loops	24	14
pma_with_sink	25	14
s27_with_loops	5	12
s27_with_sink	6	12
s298	135	8
shiftreg	8	2
tma_with_loops	20	6
tma_with_sink	21	6
train11_with_hidden_states_with_sink	11	4
train11_with_loops	9	4
train11_with_loops_with_hidden_states	9	4
train11_with_sink	10	4
train4_with_hidden_states_with_sink	6	4
train4_with_loops	4	4
train4_with_loops_with_hidden_states	5	4
train4_with_sink	5	4

TABLE 4
The names, number of states and inputs of benchmark FSMs used in the experiments.

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

Algorithm 2: The Greedy method.

Construct a Pair automaton A for $M = (S, X, Y, \delta, \lambda, s_0)$ and let $S_A \leftarrow S$, and $RS \leftarrow \epsilon$.
Construct a Breadth-First-Search tree T from A and Construct a look-up table $\mathcal{T}$ using δ and T.
while $|S_A| > 1$ **do**
  Select two states $s, s' \in S_A$.
  Find a sequence $\bar{x}$ on T such that $\delta(s, \bar{x}) = \delta(s', \bar{x})$.
  $RS \leftarrow RS.\bar{x}$.
  By using $\mathcal{T}$ compute $S_A = \delta(S_A, \bar{x})$.
return RS.

Algorithm 3: The PBF algorithm.

Let $\bar{S}$ contains N copies of set S, and $RS \leftarrow \epsilon$, $RS \leftarrow \epsilon$, $\ell \leftarrow 0$.
while ℓ *is less than a threshold* **do**
  $\ell \leftarrow \ell + 1$
  while *not all input sequences of length* ℓ *are visited* **do**
    Let $\bar{X}$ contains N defined input sequences of length ℓ also let $\bar{S}_i$ and $\bar{X}_i$ denotes the ith set and ith sequences from $\bar{S}$ and $\bar{X}$, respectively.
    Compute in parallel $\delta_i(\bar{S}_i, \bar{X}_i)$.
    if *Exists* $\delta_i(\bar{S}_i, \bar{X}_i)$ *such that* $|\delta_i(\bar{S}_i, \bar{X}_i)| == 1$ **then**
      $RS \leftarrow RS.\bar{X}_i$ and return RS.
    Select j such that $\delta_j(\bar{S}_j, \bar{X}_j)$ has minimum size.
    Let $\bar{S}$ contains N copies of $\delta_j(\bar{S}_j, \bar{X}_j)$ and also let $RS \leftarrow \bar{X}_j$

Algorithm 4: The ST algorithm.

Let $\hat{S} = \{\{S\}\}$ for $M = (S, X, Y, \delta, \lambda, s_0)$ $PDS \leftarrow \epsilon$, and Push $(\hat{S}, \epsilon)$ to queue q.
while $PDS == \epsilon$ **do**
  $(\hat{S}', \bar{x}) \leftarrow pop(q)$
  foreach $x \in X$ **do**
    $\hat{S}'' \leftarrow \hat{\delta}(\hat{S}', x)$.
    if $|\hat{S}''| = n$ **then**
      $PDS \leftarrow \bar{x}.x$.
    else
      Push $(\hat{S}'', \bar{x}.x)$ to q.
return PDS.

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 [2], [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 ϵ. 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 $(s524, s721)$, $(s70, s71)$, $(s304, s3088)$, $(s344, s2933)$, and $(s1080, s3258)$. To use the FSM in the experiments, transitions labelled with a common input symbol $I21.1$ for each of these 10 states were modified to merge at state $s524$. 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(β) to refer to our approach, where variable β 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 PDSs from FSMs, which is a classic algorithm described in detail in Algorithm 4. During experiments we could not use the tabular $\mathcal{Q}$-learning algorithm due to space limitations. For example for the smallest FSM subject, which has 32 states and two inputs, the tabular $\mathcal{Q}$-learning framework would require a 2^{32} by 2 table leading to $8,589,934,592$ entries each holding two (for holding $\mathcal{Q}$ 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 $\mathcal{Q}$-Graph algorithm, we noted the collected rewards and the $\mathcal{Q}$-nodes and edges information of the $\mathcal{Q}$-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 $\mathcal{Q}$-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 result 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 $n > 100$. 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.

Summary

The proposed framework is 100 times faster, requiring 66 times less memory, but generated PDSs are 12% longer than those of the benchmark (ST) algorithm.

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.

The results are promising. During experiments, we observed that the ST algorithm could only process 20% of the FSMs with less than four inputs and a maximum of 2048 states. The ST could not process other FSMs. On the other hand, the Q(PDS) algorithm could process all the FSMs and compute the PDSs when $n < 16843$. This implies that roughly the ST algorithm could process 240 FSMs out of 4000 (6% of the FSMs) and Q(PDS) could process 3600 (90% of the FSMs). So the $\mathcal{Q}$-Graph framework increased the scalability by 1400% [5]. Statistical analysis supports these findings: generally, the Wilcoxon hypothesis test rejects the null hypothesis, and Cohen's d metric suggests a strong difference.

We compared the ST and the Q(PDS) algorithms with respect to test subjects that the ST could process. Considering the statistical analyses and Figure 8, we conclude that the Q(PDS) algorithm is slightly faster than the ST algorithm (26% on average), which is different from the results observed in complete FSMs.

With respect to memory requirements (see Figures 6 and 9), we observe that comparing the results obtained from complete FSMs, the memory requirements of the algorithms increased. This is because finding a PDS from partial machines requires more processing. As we will see, the lengths of the PDSs derived from partial FSMs are much longer. This is expected as there are fewer inputs to construct a PDS. This consequently increased the memory requirements of the algorithms. Therefore, the Q(PDS) algorithm generated relatively large $\mathcal{Q}$-Graphs. However, as can be seen from the figures and statistical studies, the proposed algorithm requires less memory space than the ST algorithm (40% on average).

5.3.2 Length of sequences

When analysing the lengths of the PDSs (Figure 10), we observe that the ST and the Q(PDS) algorithms computed PDSs with similar lengths. However, the Wilcoxon hypothesis test rejects the null hypothesis and Cohen's d metric shows strong difference. So we conclude that Q(PDS) generates longer PDSs from partial FSMs (maximum was 24% longer and $4\% - 8\%$ longer on average). This is a promising result, and further investigation revealed that this was again due to the lack of defined inputs. The lack of useful inputs resulted in fewer options for algorithms, and both ended up computing similar PDSs.

Summary

The state-of-the-art approach (ST algorithm) failed to process most test subjects. The $\mathcal{Q}$-Graph framework increases the scalability by 1400%. The ST and the $\mathcal{Q}$-Graph approach requires 40% less memory, and the $\mathcal{Q}$-Graph is 26% faster.

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

5. $((3600 - 240)/240) * 100 = (3360/240) * 100 = 1400\%$

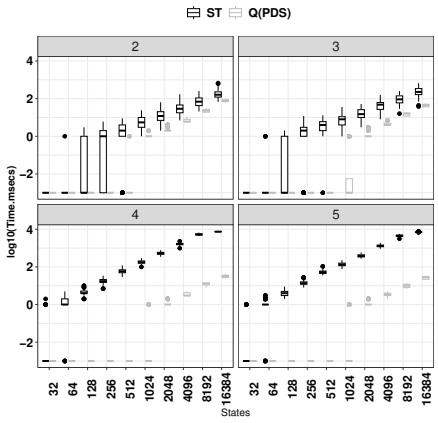

Fig. 5. The average time required to construct PDSs from complete randomly generated FSMs.

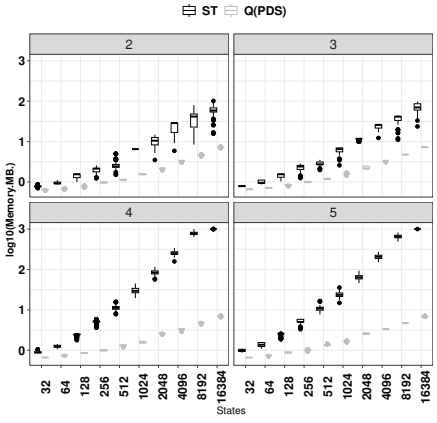

Fig. 6. The average memory required to construct PDSs from complete randomly generated FSMs.

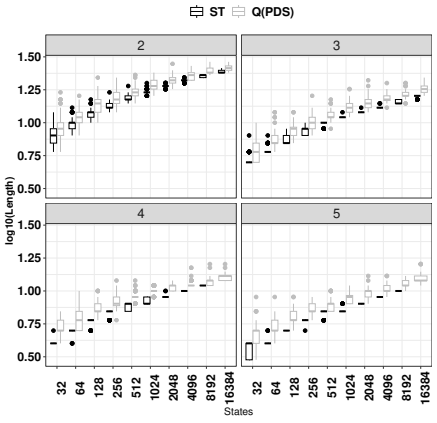

Fig. 7. The average lengths of PDSs derived from complete randomly generated FSMs.

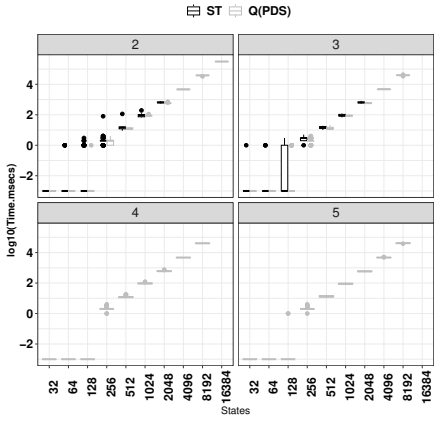

Fig. 8. The average time required to construct PDSs from partial randomly generated FSMs.

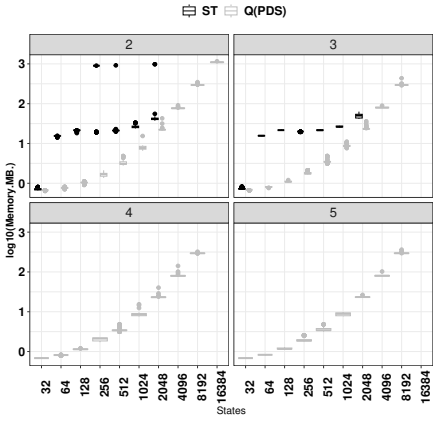

Fig. 9. The average memory required to construct PDSs from partial randomly generated FSMs.

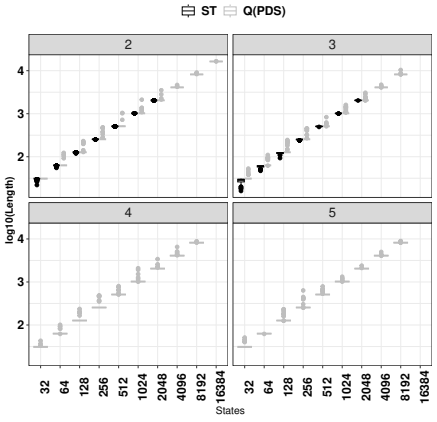

Fig. 10. The average lengths of PDSs derived from partial randomly generated FSMs.

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 (the Wilcoxon and Cohen's d tests support this claim). Specifically, the proposed algorithm used 166 times 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.

Summary

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 log_{10} 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.

Spec(n,i)	Complete -PDS Wilcoxon(Q(PDS),ST)			Partial - PDS Wilcoxon(Q(PDS),ST)			Complete - RS Wilcoxon(Q(RS),GREEDY)			Partial - RS Wilcoxon(Q(RS),PBF)		
	Length	Time	Memory	Length	Time	Memory	Length	Time	Memory	Length	Time	Memory
(32, 2)	5^{-35}	1^{-21}	1^{-34}	2^{-42}	1^{-38}	1^{-35}	1^{-34}	1^{-1}	1^{-34}	8^{-36}	1^{-1}	7^{-36}
(32, 3)	3^{-36}	1^{-22}	6^{-35}	3^{-36}	1	3^{-35}	1^{-34}	1	4^{-35}	1^{-35}	1	4^{-36}
(32, 4)	1^{-36}	4^{-3}	1^{-35}	NaN	NaN	NaN	1^{-34}	3^{-1}	2^{-36}	1^{-35}	7^{-2}	2^{-36}
(32, 5)	9^{-36}	6^{-3}	1^{-35}	NaN	NaN	NaN	1^{-34}	1	3^{-35}	1^{-35}	9^{-2}	2^{-36}
(64, 2)	1	3^{-35}	4^{-35}	3^{-41}	3^{-2}	1^{-34}	1^{-34}	3^{-24}	2^{-35}	5^{-36}	3^{-2}	1^{-35}
(64, 3)	1^{-1}	7^{-36}	1^{-34}	4^{-36}	1^{-1}	3^{-35}	1^{-34}	1^{-23}	4^{-35}	8^{-36}	2^{-2}	7^{-36}
(64, 4)	6^{-37}	1^{-18}	1^{-35}	NaN	NaN	NaN	1^{-34}	4^{-23}	2^{-36}	9^{-36}	1^{-3}	9^{-36}
(64, 5)	4^{-36}	2^{-19}	1^{-35}	NaN	NaN	NaN	1^{-34}	3^{-21}	2^{-35}	1^{-35}	1^{-2}	9^{-36}
(128, 2)	4^{-35}	5^{-10}	8^{-35}	4^{-40}	5^{-5}	9^{-35}	1^{-34}	2^{-23}	6^{-35}	4^{-36}	8^{-20}	1^{-35}
(128, 3)	2^{-35}	5^{-7}	7^{-35}	8^{-36}	5^{-10}	1^{-35}	1^{-34}	1^{-21}	1^{-34}	9^{-36}	5^{-19}	5^{-36}
(128, 4)	3^{-36}	2^{-18}	9^{-35}	NaN	NaN	NaN	1^{-34}	5^{-20}	1^{-34}	8^{-36}	8^{-19}	9^{-36}
(128, 5)	3^{-36}	2^{-18}	4^{-35}	NaN	NaN	NaN	1^{-34}	3^{-20}	1^{-34}	1^{-35}	2^{-19}	3^{-36}
(256, 2)	4^{-35}	9^{-12}	1^{-34}	6^{-40}	1^{-36}	7^{-35}	1^{-34}	2^{-34}	1^{-34}	2^{-36}	8^{-28}	9^{-36}
(256, 3)	3^{-35}	9^{-16}	1^{-34}	5^{-36}	2^{-36}	4^{-35}	1^{-34}	1^{-27}	1^{-34}	1^{-35}	2^{-24}	1^{-36}
(256, 4)	5^{-36}	3^{-18}	1^{-34}	NaN	NaN	NaN	1^{-34}	5^{-25}	5^{-35}	9^{-36}	1^{-35}	1^{-36}
(256, 5)	1^{-36}	3^{-18}	9^{-35}	NaN	NaN	NaN	1^{-34}	2^{-25}	9^{-35}	5^{-36}	3^{-34}	2^{-36}
(512, 2)	3^{-35}	2^{-16}	1^{-34}	1^{-40}	9^{-35}	1^{-34}	1^{-34}	1^{-34}	1	3^{-36}	2^{-35}	1^{-35}
(512, 3)	5^{-36}	6^{-18}	1^{-34}	4^{-36}	9^{-35}	7^{-35}	1^{-34}	1^{-34}	7^{-35}	5^{-36}	3^{-35}	5^{-36}
(512, 4)	2^{-35}	3^{-18}	1^{-34}	NaN	NaN	NaN	1^{-34}	1^{-34}	9^{-35}	5^{-36}	7^{-35}	2^{-36}
(512, 5)	2^{-36}	3^{-18}	1^{-34}	NaN	NaN	NaN	1^{-34}	9^{-35}	7^{-35}	4^{-36}	3^{-35}	6^{-36}
(1024, 2)	1^{-35}	8^{-36}	1^{-35}	1^{-41}	1^{-34}	1^{-34}	NaN	NaN	NaN	2^{-36}	7^{-35}	8^{-36}
(1024, 3)	6^{-36}	2^{-22}	4^{-35}	9^{-36}	1^{-34}	1^{-34}	NaN	NaN	NaN	3^{-36}	7^{-35}	5^{-36}
(1024, 4)	2^{-35}	2^{-20}	1^{-34}	NaN	NaN	NaN	NaN	NaN	NaN	3^{-36}	3^{-35}	2^{-36}
(1024, 5)	1^{-35}	8^{-21}	3^{-35}	NaN	NaN	NaN	NaN	NaN	NaN	3^{-36}	7^{-35}	9^{-37}
(2048, 2)	7^{-36}	3^{-35}	1^{-35}	8^{-42}	1^{-34}	1^{-34}	NaN	NaN	NaN	2^{-36}	1^{-34}	1^{-35}
(2048, 3)	7^{-36}	4^{-35}	3^{-35}	9^{-36}	1^{-34}	1^{-34}	NaN	NaN	NaN	4^{-36}	1^{-34}	1^{-35}
(2048, 4)	8^{-36}	3^{-35}	8^{-35}	NaN	NaN	NaN	NaN	NaN	NaN	2^{-36}	8^{-35}	1^{-35}
(2048, 5)	7^{-36}	4^{-35}	8^{-35}	NaN	NaN	NaN	NaN	NaN	NaN	3^{-36}	4^{-35}	1^{-35}
(4096, 2)	6^{-36}	1^{-34}	7^{-36}	NaN	NaN	NaN	NaN	NaN	NaN	1^{-36}	1^{-34}	1^{-35}
(4096, 3)	1^{-35}	5^{-35}	8^{-36}	NaN	NaN	NaN	NaN	NaN	NaN	3^{-36}	1^{-34}	1^{-35}
(4096, 4)	3^{-36}	6^{-35}	1^{-35}	NaN	NaN	NaN	NaN	NaN	NaN	2^{-36}	1^{-34}	1^{-35}
(4096, 5)	6^{-36}	1^{-34}	1^{-35}	NaN	NaN	NaN	NaN	NaN	NaN	2^{-36}	1^{-34}	1^{-35}
(8192, 2)	1^{-35}	1^{-34}	6^{-35}	NaN	NaN	NaN	NaN	NaN	NaN	9^{-29}	1^{-26}	1^{-28}
(8192, 3)	4^{-35}	1^{-34}	1^{-35}	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(8192, 4)	6^{-38}	1^{-34}	1^{-35}	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(8192, 5)	4^{-36}	1^{-34}	1^{-35}	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(16384, 2)	1^{-35}	1^{-34}	1^{-34}	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(16384, 3)	1^{-35}	1^{-34}	8^{-35}	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(16384, 4)	4^{-19}	1^{-34}	1^{-34}	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(16384, 5)	5^{-19}	1^{-34}	1^{-34}	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

TABLE 5
Wilcoxon hypothesis test where the null hypothesis is *two populations are same* and is accepted if $p > 0.05$. The red coloured text indicates where the hypothesis is accepted. The black-coloured text indicates cases for which the Wilcoxon test rejects the null hypothesis, and NaN indicates the cases for which the state-of-the-art algorithms fail to produce results.

Spec(n,i)	Complete -PDS Cohen's D(Q(PDS),ST)			Partial - PDS Cohen's D(Q(PDS),ST)			Complete - RS Cohen's D(Q(RS),GREEDY)			Partial - RS Cohen's D(Q(RS),PBF)		
	Length	Time	Memory	Length	Time	Memory	Length	Time	Memory	Length	Time	Memory
(32, 2)	0.369	0.522	0.270	0.844	0.331	0.661	0.898	0.069	0.468	0.343	0.059	0.261
(32, 3)	0.542	0.512	0.768	0.134	0.04	0.643	0.646	0.103	0.290	0.382	0.022	0.836
(32, 4)	0.499	0.722	0.247	NaN	NaN	NaN	0.311	0.013	0.778	0.758	0.105	0.551
(32, 5)	0.099	0.253	0.105	NaN	NaN	NaN	0.099	0.1373	0.634	0.844	0.136	0.817
(64, 2)	0.103	0.345	0.760	0.877	0.220	0.615	0.235	0.343	0.337	0.252	0.221	0.779
(64, 3)	0.133	0.325	0.847	0.561	0.015	0.585	0.261	0.724	0.292	0.581	0.417	0.607
(64, 4)	0.667	0.536	0.858	NaN	NaN	NaN	0.265	0.888	0.849	0.220	0.653	0.859
(64, 5)	0.811	0.400	0.385	NaN	NaN	NaN	0.776	0.532	0.497	0.791	0.615	0.279
(128, 2)	0.226	0.661	0.346	0.491	0.333	0.560	0.433	0.769	0.534	0.502	0.282	0.645
(128, 3)	0.323	0.751	0.386	0.388	0.751	0.819	0.399	0.891	0.372	0.713	0.236	0.631
(128, 4)	0.827	0.890	0.577	NaN	NaN	NaN	0.724	0.638	0.810	0.569	0.422	0.867
(128, 5)	0.362	0.455	0.625	NaN	NaN	NaN	0.707	0.392	0.321	0.509	0.285	0.577
(256, 2)	0.834	0.647	0.848	0.515	0.381	0.488	0.780	0.768	0.713	0.769	0.876	0.609
(256, 3)	0.633	0.377	0.846	0.522	0.597	0.370	0.390	0.492	0.506	0.388	0.828	0.852
(256, 4)	0.421	0.553	0.515	NaN	NaN	NaN	0.497	0.823	0.716	0.386	0.595	0.388
(256, 5)	0.520	0.784	0.572	NaN	NaN	NaN	0.518	0.345	0.505	0.656	0.704	0.333
(512, 2)	0.407	0.818	0.402	0.575	0.642	0.638	0.710	0.624	0.782	0.520	0.443	0.850
(512, 3)	0.453	0.713	0.795	0.716	0.640	0.705	0.852	0.587	0.778	0.581	0.865	0.616
(512, 4)	0.631	0.732	0.551	NaN	NaN	NaN	0.810	0.500	0.754	0.718	0.677	0.400
(512, 5)	0.670	0.583	0.477	NaN	NaN	NaN	0.512	0.484	0.494	1.866	0.600	0.611
(1024, 2)	0.622	0.456	0.729	0.590	0.839	0.768	NaN	NaN	NaN	0.437	0.787	0.639
(1024, 3)	0.614	0.648	0.566	0.852	0.756	0.799	NaN	NaN	NaN	0.899	0.503	0.647
(1024, 4)	0.749	0.685	0.745	NaN	NaN	NaN	NaN	NaN	NaN	0.501	0.813	0.723
(1024, 5)	0.783	0.899	0.819	NaN	NaN	NaN	NaN	NaN	NaN	0.792	0.565	0.669
(2048, 2)	0.681	0.724	0.699	0.791	0.891	0.706	NaN	NaN	NaN	0.808	0.557	0.688
(2048, 3)	0.614	0.804	0.659	0.736	0.708	0.690	NaN	NaN	NaN	0.538	0.549	0.828
(2048, 4)	0.610	0.620	0.672	NaN	NaN	NaN	NaN	NaN	NaN	0.872	0.886	0.891
(2048, 5)	0.729	0.742	0.878	NaN	NaN	NaN	NaN	NaN	NaN	0.844	0.580	0.823
(4096, 2)	0.884	0.867	0.749	NaN	NaN	NaN	NaN	NaN	NaN	0.796	0.800	0.719
(4096, 3)	0.736	0.829	0.858	NaN	NaN	NaN	NaN	NaN	NaN	0.656	0.623	0.852
(4096, 4)	0.748	0.801	0.670	NaN	NaN	NaN	NaN	NaN	NaN	0.761	0.717	0.741
(4096, 5)	0.697	0.848	0.861	NaN	NaN	NaN	NaN	NaN	NaN	0.657	0.853	0.661
(8192, 2)	0.724	0.745	0.816	NaN	NaN	NaN	NaN	NaN	NaN	0.846	0.852	0.760
(8192, 3)	0.826	0.738	0.832	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(8192, 4)	0.869	0.753	0.828	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(8192, 5)	0.780	0.865	0.773	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(16384, 2)	0.784	0.807	0.847	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(16384, 3)	0.891	0.813	0.826	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(16384, 4)	0.873	0.887	0.897	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(16384, 5)	0.865	0.856	0.837	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

TABLE 6
Cohen's D strength test. A value less than 0.2 indicates weak strength, and a value larger than 0.5 indicates strong strength. The red-coloured text indicates where we observe weak strength.

Summary

The state-of-the-art approach (PBF algorithm) and the Q-Graph framework require similar memory space while computing RSs. The Q-Graph framework is 12 times faster, but, on average, the lengths of the RSs are 33% longer than the RSs produced by the PBF algorithm.

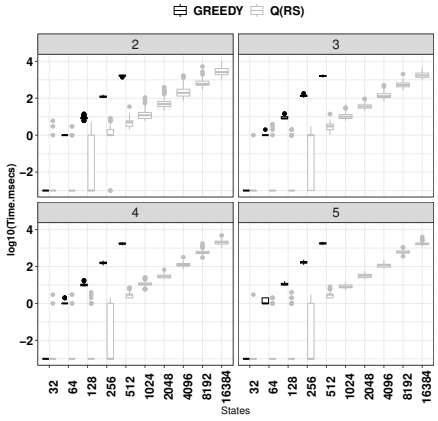

Fig. 11. The average time required to construct RSs from complete randomly generated FSMs.

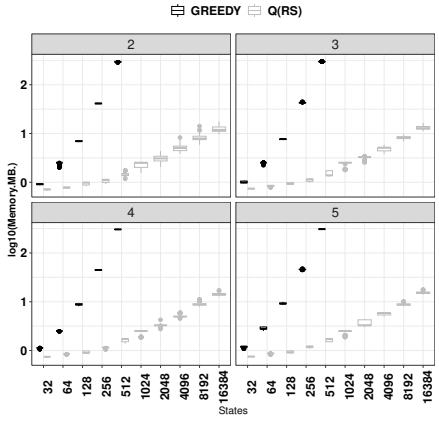

Fig. 12. The average memory required to construct RSs from complete randomly generated FSMs.

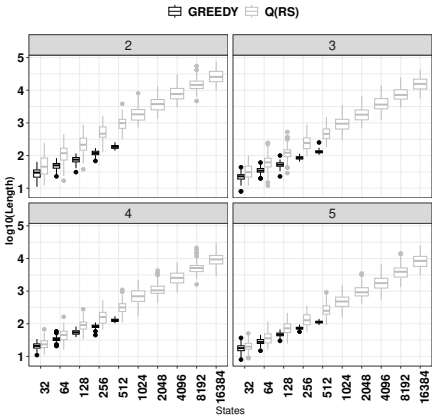

Fig. 13. The average lengths of RSs derived from complete randomly generated FSMs.

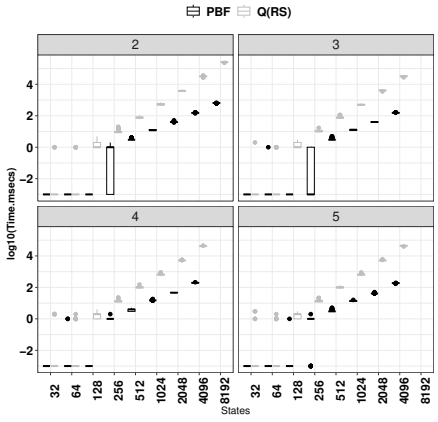

Fig. 14. The average time required to construct RSs from partial randomly generated FSMs.

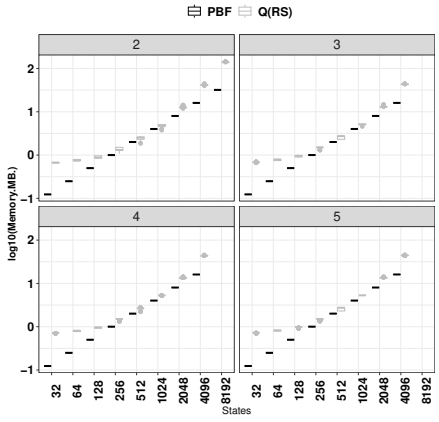

Fig. 15. The average memory required to construct RSs from partial randomly generated FSMs.

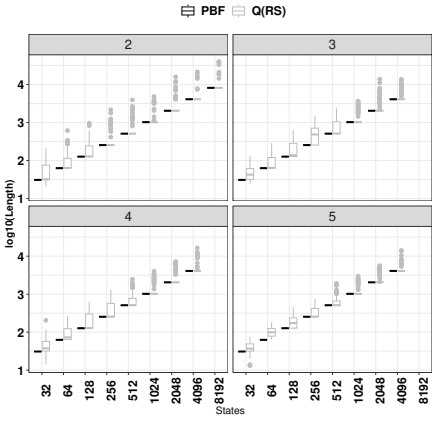

Fig. 16. The average lengths of RSs derived from partial randomly generated FSMs.

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

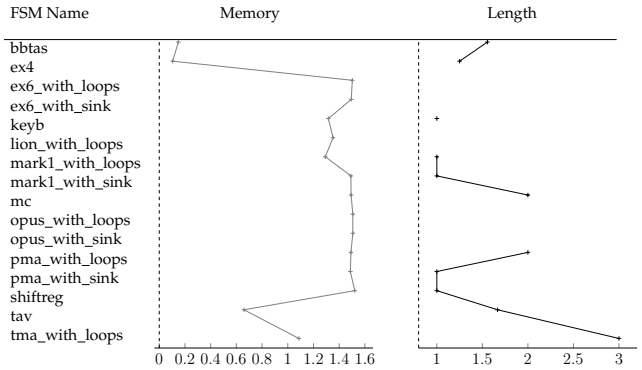

TABLE 7
PDS results of benchmark experiments.

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.

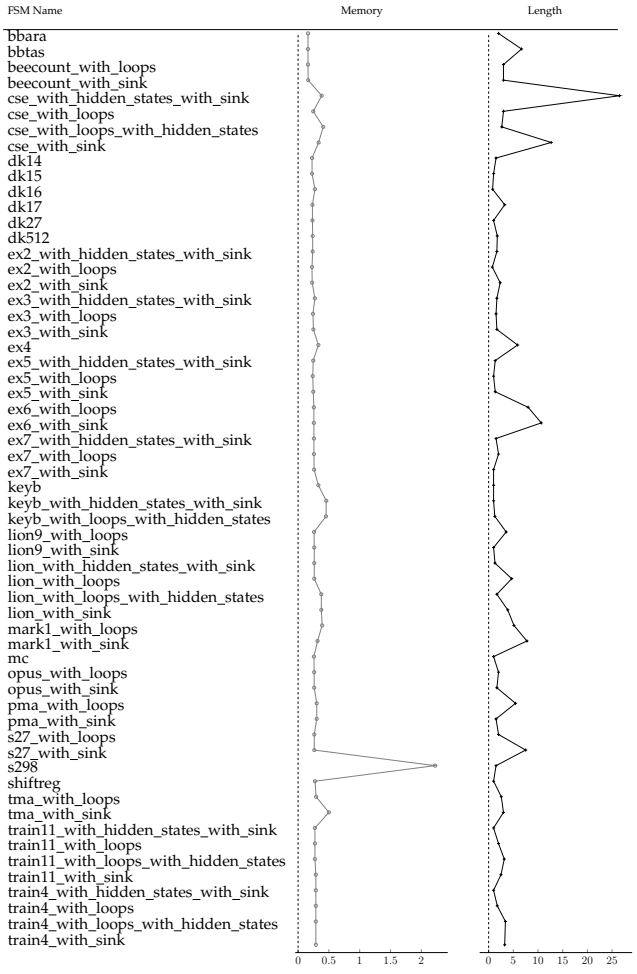

TABLE 8
RS results of benchmark experiments.

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 $\mathcal{Q}$-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 (tFSM) $tM = (S, X, Y, \delta, \lambda, s_0, \Delta)$ is an FSM with a single clock. It has a finite set of states (S), inputs (X) and outputs (Y). The δ and λ 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* (ℓ). 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 $\Delta : S \to S \times (\mathbb{N} \cup \{\infty\})$ maps a

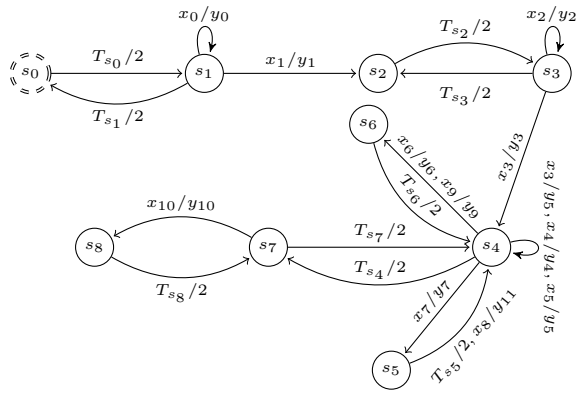

Fig. 17. A timed FSM models the behaviour of passive infrared motion sensor(PIR).

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 first must 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 $\delta(s, x) = s'$, $\lambda(s, x) = y$, 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_5, and the location is $\ell(s_5, 1)$. If input x_9 is applied before tM moves to $\ell(s_5, 2)$ then tM updates its state and location to s_4 and $\ell(s_4, 0)$ producing output y_{12}. However, if no input is applied, the state and location will be updated to s_4 and $\ell(s_4, 0)$ 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* $\chi : S \times \mathbb{N} \times \mathbb{N} \to S \times \mathbb{N}$. Given a location $\ell(s, t_s)$, the function χ specifies the next location that the tFSM will reach after waiting some time t' (*time input*) from a location. In the following let $\Delta(s) = (s', T_s)$.

Symbols	Definitions
tM, Δ	A timed FSM, Timed transition function.
ℓ, $\mathcal{L}$, $\hat{\mathcal{L}}$, $\mathcal{L}^0$	A location, a set of locations, a set of sets of locations, the set of all locations.
t_s, T_s, r_s	Local time, local-time limit, remaining time.
χ, $\hat{\chi}$	The timed next location function of a location (or a set of locations), the next set of sets of locations function.
$\mathbb{U}$, $\hat{\mathbb{U}}$	The next location function of a location (or a set of locations), next location function for a set of sets of locations.
$\mathbb{L}$, $\hat{\mathbb{L}}$	Output function of location (or a set of locations), output function of a set of sets of locations.

TABLE 9
List of symbols regarding timed FSMs.

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

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

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

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. $\bar{x}_t \in (X \cup \mathbb{N})^*$. For example, consider tFSM tM in Figure 17. Assume we want to reach location $\ell(s_6, 0)$ from $\ell(s_5, 1)$. This can be done by waiting 1 unit of time, switching to state s_4, and then applying input x_6. This defines timed input sequence $\bar{x}_t = 1x_6$. When executing these transitions, tM follows $\ell(s_5, 1)\ell(s_4, 0)\ell(s_6, 0)$ (please note $T_{s_5} = 2$ and $\Delta(s_5) = (s_4, 2)$). However, if the input sequence is $\bar{x}_t = x_8x_6$, then tM reaches $\ell(s_6, 0)$ from $\ell(s_5, 0)$ and produce the output $y_{11}y_6$. 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 δ, λ, and χ return ε.

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

$$\mathbb{U}(\ell(s, t_s), k) = \begin{cases} \ell(\delta(s, k), 0), \text{ iff } k \in X, \\ \chi(\ell(s, t_s), k), \text{ iff } k \in \mathbb{N} \end{cases}$$

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

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

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

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

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 $\bar{x}_t$ is a timed reset sequence (tRS) for tM iff $|\mathbb{U}(\mathcal{L}^0, \bar{x})| = 1$.

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 $\bar{x}_t = x_1 2\ x_2 x_1 2 x_3 x_{10} 2\ 2$ is a tRS for the tFSM given in Figure 17, $\mathbb{U}(\ell(s_0,0), x_1 2 x_2 x_1 2 x_3 x_{10} 2\ 2) = \ell(s_4,0)$,$\mathbb{U}(\ell(s_1,0), x_1 2 x_2 x_1 2 x_4 3 x_{10} 2\ 2) = \ell(s_4,0)$, and $\mathbb{U}(\ell(s_2,0), x_1 2 x_3 x_1 2 x_3 x_{10} 2\ 2) = \ell(s_4,0)$.

We represent the tRS construction problem as an MDP $P = (\mathcal{Z}, \mathcal{A}, \mathcal{P}_t, R_{tRS})$ in the following way. Let tM be a tFSM, and we consider a set of locations $\mathcal{L}$ as a state $\mathcal{K} \in \mathcal{Z}$ of the MDP P, and therefore the set of possible states of the MDP P is the set $pow(S) \times r_S$, where r_S is the minimum remaining time observed in set S. The set of actions of P is defined using the set of inputs and non-negative numbers not exceeding r_S, i.e., $X \cup \{1, 2, \ldots, r_S\}$. For example, we introduced action a_{x_1} to $\mathcal{A}$ for $x_1 \in X$ and a_2 to $\mathcal{A}$ 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 $X \cup \{1, 2, \ldots, r_S\}$ of the tFSM tM to the corresponding action $a \in \mathcal{A}$ of the MDP P and vice versa, and we let $st()$ map a set of locations $\mathcal{L}' \in pow(S) \times r_S$ of the tFSM tM to the corresponding state $\mathcal{K}' \in \mathcal{Z}$ of the MDP P and vice versa. A state change from $\mathcal{K}$ to $\mathcal{K}'$ is triggered by an application of an action $a \in \mathcal{A}$. We define the probability function $\mathcal{P}_t(\mathcal{K}'|\mathcal{K}, a)$ as follows, let $\mathbb{U}(st(\mathcal{K}), i(a)) = st(\mathcal{K}')$.

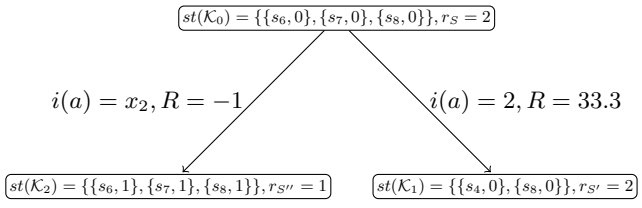

Fig. 18. The execution of the reward function for locations $\{s_6,0\},\{s_7,0\},\{s_8,0\}$ of tFSM given in Figure 17. Note applying minimum remaining time causes locations $\{s_6,0\},\{s_7,0\}$ to merge at location $\{s_4,0\}$.

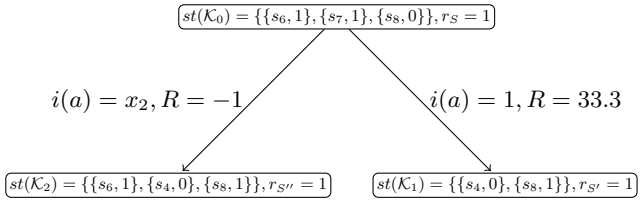

Fig. 19. The execution of the reward function for locations $\{s_6,1\},\{s_7,1\},\{s_8,0\}$ of tFSM given in Figure 17. Note applying minimum remaining time causes locations $\{s_6,1\},\{s_7,1\}$ reaching location $\{s_4,0\}$.

$$\mathcal{P}_t(\mathcal{K}'|\mathcal{K},a) = \begin{cases} 1, \text{ iff } \mathbb{U}(st(\mathcal{K}), i(a)) = st(\mathcal{K}'), \\ 0, \text{ otherwise.} \end{cases}$$

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 $\mathcal{K}$ associated with $\mathcal{L}^0$ to another MDP state that is associated with $\mathcal{L}$ such that $|\mathcal{L}| = 1$. If input given by $i(a)$ is not defined for the set of locations given by $st(\mathcal{K})$, then the reward function returns NAN. For the defined actions, the reward function to compute a tRS is similar to that of RS given in Section 4.2 and uses ϕ. In Figure 18, we demonstrate this process for tFSM given in Figure 17. However, we also consider the local time differences between the locations. Let $\psi(\mathcal{L}) = \Sigma_{\ell(s,t_s),\ell(s',t_{s'})\in\mathcal{L}'} abs((t_s) - (t_{s'}))$ be the sum of the differences of local times of the locations in set $\mathcal{L}$. The intuition behind ψ is to evaluate the impact of actions with respect to time. As the timed reset sequence resets a set of locations, an action that leads to divergence in local times must be penalised but awarded if it leads to convergence. In Figure 19, we demonstrate this process for tFSM given in Figure 17.

$$R_{tRS}(\mathcal{K},a) = \begin{cases} -1, & \text{if } |st(\mathcal{K})| = |st(\mathcal{K}')|, \\ NAN, & \text{if } i(a) \text{ is undefined for } st(\mathcal{K}), \\ \dfrac{100 * \phi}{|st(\mathcal{K})|}, & \text{if } \psi(st(\mathcal{K}')) = 0, \\ \dfrac{100\phi}{|st(\mathcal{K})| * \psi(st(\mathcal{K}'))} & \text{otherwise.} \end{cases}$$

In implementing the tRS problem using $\mathcal{Q}$-Graph, the initial node of the $\mathcal{Q}$-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 $\bar{x}_t$ is a timed preset distinguishing

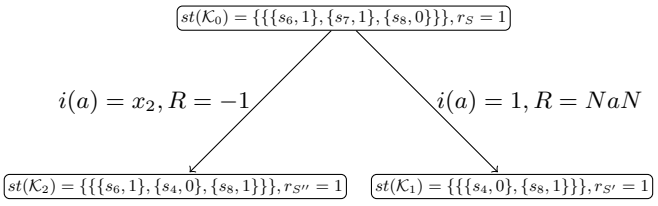

Fig. 20. The execution of the reward function for locations $\{s_6, 1\}, \{s_7, 1\}, \{s_8, 0\}$ of tFSM given in Figure 17. Note applying minimum remaining time causes locations $\{s_6, 1\}, \{s_7, 1\}$ reaching location $\{s_4, 0\}$.

sequence (tPDS) for tM if for every pair of s and s' of distinct states of tM and every pair of locations (ℓ, ℓ') where $\ell \in \mathcal{L}_s$ and $\ell' \in \mathcal{L}_{s'}$, we have, $\mathbb{L}(\ell, \bar{x}_t) \neq \mathbb{L}(\ell', \bar{x}_t)$.

For example, the timed input sequence $\bar{x}_t = x_0x_2$ $x_3x_8x_{10}2x_0x_3x_{10}$ 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 $\mathbb{L}(\ell(s_0, l_{s_0}), \bar{x}_t) = NNNNNy_1NN$, $\mathbb{L}(\ell(s_1, l_{s_1}), \bar{x}) = y_1NNNNy_1NN$, and for example $\mathbb{L}(\ell(s_2, l_{s_2}), \bar{x}) = NNN$ $NNNy_3N$.

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

Again by abusing notations introduced in Sections 4.2, and 6.3, we use $i(.)$ to convert input from set $X \cup \{1, 2, \ldots, r_{\hat{S}}\}$ to an action $a \in \mathcal{A}$ 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 $\hat{\mathcal{L}}$ of $tFSM$ to a state $\mathcal{K}$ of P, and vice-versa.

The state transitions of P are given as follows.

$$\mathcal{P}_t(\mathcal{K}'|\mathcal{K}, a) = \begin{cases} 1, \text{ iff } \hat{\mathbb{U}}(\hat{st}(\mathcal{K}), i(a)) = st(\mathcal{K}'), \\ 0, \text{ otherwise.} \end{cases}$$

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_{tPDS} 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 $\mathcal{Q}$-Graph, the initial node has $st(\mathcal{L}^0)$.

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 ∞. 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 FSMs. To study tRSs, we could only create one hundred tFSMs for each n, i value from $\{32, 64\}$ and $\{3, 4, 5\}$[6]. 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 FSMs (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 $\mathcal{Q}$-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.

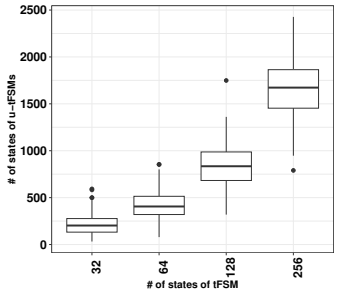

Fig. 21. The average state sizes of unfolded FSMs vs states sizes of folded tFSMs used in tPDS generation.

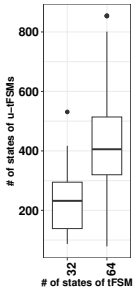

Fig. 22. The average state sizes of unfolded FSMs vs states sizes of folded tFSMs used in tRS generation.

During the experiments, we fed the algorithms with the tFSMs and noted the time, the memory required to construct sequences, and the length of the sequences.

6. As the number of states increases, finding a tFSM having a tRS becomes a hard task by using the random FSM generator.

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 $\mathcal{Q}$ 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 $\mathcal{Q}$-graph approach, and the $\mathcal{Q}$-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 $\mathcal{Q}$-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 $\mathcal{Q}$-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 $\mathcal{Q}$-Graph framework avoids constructing the $\mathcal{Q}$-table, which is the fundamental data structure of tabular $\mathcal{Q}$-learning framework. In comparison, the $\mathcal{Q}$-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 $2^{16384} * 5$ 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]

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.

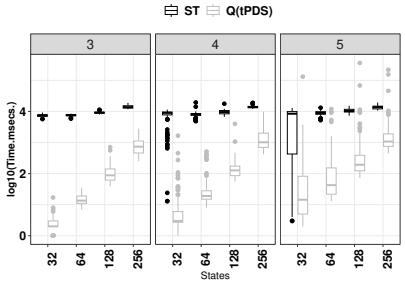

Fig. 23. The average time required to construct tPDSs from partial randomly generated tFSMs.

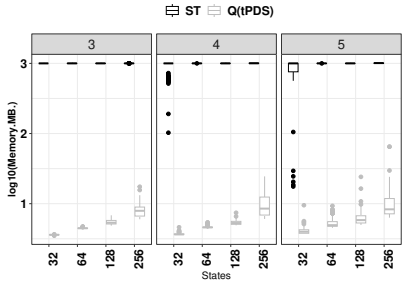

Fig. 24. The average memory required to construct tPDSs from partial randomly generated tFSMs.

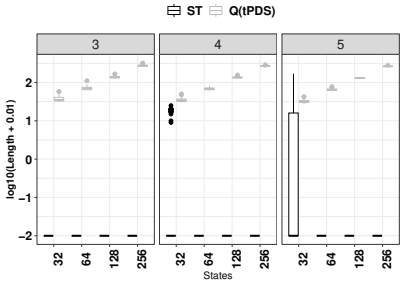

Fig. 25. The average lengths of tPDSs constructed from partial randomly generated tFSMs.

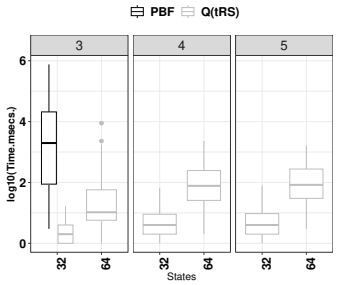

Fig. 26. The average time required to construct tRSs from partial randomly generated tFSMs.

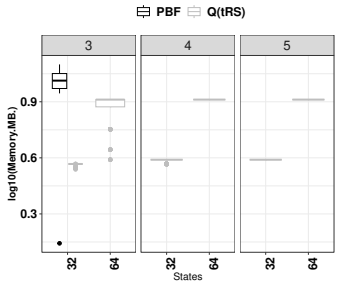

Fig. 27. The average memory required to construct tRSs from partial randomly generated tFSMs.

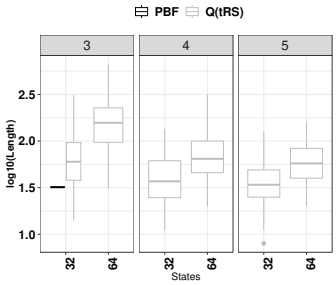

Fig. 28. The average lengths of tRSs constructed from partial randomly generated tFSMs.

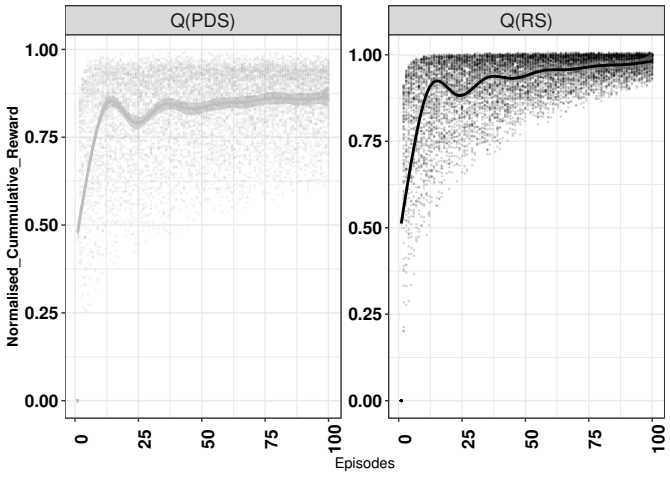

Fig. 29. Normalised cumulative rewards collected by $\mathcal{Q}$-Graph framework while constructing RSs and PDSs from a given FSM.

this risk. We also took into account the possibility of bias introduced during the generation of syntactic FSMs, which could pose a potential threat. To mitigate this, we employed a widely-used tool that has been extensively utilised in similar research endeavours. Additionally, we acknowledge the potential for implementation errors in our approach. To address this concern, for each FSM M (or tFSM tM), we applied the generated RS (or PDS) sequence to M (or tM) to check if the sequence was really an RS (or PDS) for M (or tM).

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 $\mathcal{Q}$-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. Uraz Cengiz Türker has been partially supported by the Security Lancaster, IRL1032 Poison Attack Mitigation grant. Khaled El-Fakih 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 (RoaRQ), Investigation 009 Model-based monitoring and calibration of quantum computations (ModeMCQ), grant reference EP/W032635/1, as well as the King's Quantum grant provided by King's College London.

REFERENCES

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

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

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

[4] R. Dorofeeva, K. El-Fakih, and N. Yevtushenko, "An improved conformance testing method," in *FORTE*, 2005, pp. 204–218.

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

[6] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes, "Generating finite state machines from abstract state machines," in *Proceedings of the ACM SIGSOFT Symposium on Software Testing and Analysis*, 2002, pp. 112–122.

[7] P. Fiterau-Brostean, B. Jonsson, K. Sagonas, and F. Tåquist, "Automata-based automated detection of state machine bugs in protocol implementations." in *NDSS*, 2023.

[8] E. Muškardin, B. K. Aichernig, I. Pill, A. Pferscher, and M. Tappler, "Aalpy: An active automata learning library," *Innov. Syst. Softw. Eng.*, vol. 18, no. 3, p. 417–426, sep 2022. [Online]. Available: https://doi.org/10.1007/s11334-022-00449-3

[9] F. Ipate, "Learning finite cover automata from queries," *Journal of Computer and System Sciences*, vol. 78, no. 1, pp. 221–244, 2012, jCSS Knowledge Representation and Reasoning. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S002200001100047X

[10] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0890540187900526

[11] 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.

[12] 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.

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

[14] ——, "Parallel algorithms for generating distinguishing sequences for observable non-deterministic FSMs," *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 1, pp. 5:1–5:34, 2017.

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

[16] 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.

[17] 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

[18] N. E. Saraç, Ömer Faruk Altun, K. T. Atam, S. Karahoda, K. Kaya, and H. Yenigün, "Boosting expensive synchronizing heuristics," *Expert Systems with Applications*, vol. 167, p. 114203, 2021.

[19] 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.

[20] M. Lukac and K. El-Fakih, "On distinguishing sequences of several classes of reversible finite state machines," in *2021 IEEE 51st International Symposium on Multiple-Valued Logic (ISMVL)*, 2021, pp. 113–119.

[21] B. Elghadyry, F. Ouardi, Z. Lotfi, and S. Verel, "Efficient parallel derivation of short distinguishing sequences for nondeterministic finite state machines using mapreduce," *Journal of Big Data*, vol. 8, 11 2021.

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

[23] D. Damasceno, M. R. Mousavi, and A. Simão, "Learning by sampling: learning behavioral family models from software product lines," *Empir. Softw. Eng.*, vol. 26, no. 1, p. 4, 2021.

[24] R. M. Hierons, G.-V. Jourdan, H. Ural, and H. Yenigun, "Checking sequence construction using adaptive and preset distinguishing sequences," in *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*. IEEE Computer Society, 2009, pp. 157–166.

[25] 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

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

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

[28] R. M. Hierons and H. Ural, "Optimizing the length of checking sequences," *IEEE Transactions on Computers*, vol. 55, no. 5, pp. 618–629, 2006.

[29] G. V. Jourdan, H. Ural, H. Yenigun, and J. Zhang, "Lower bounds on lengths of checking sequences," *Formal Aspects of Computing*, vol. 22, no. 6, pp. 667–679, 2010.

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

[31] M. G. Merayo, M. Núñez, and I. Rodríguez, "Formal testing from timed finite state machines," *Comput. Netw.*, vol. 52, no. 2, p. 432–460, Feb. 2008.

[32] R. M. Hierons, M. García Merayo, and M. Nuñez García, "Testing from a stochastic timed system with a fault model." *Journal of Logic and Algebraic Programming*, vol. 78, no. 2, pp. 98–115, January 2009.

[33] D. Bresolin, K. El-Fakih, T. Villa, and N. Yevtushenko, "Equivalence checking and intersection of deterministic timed finite state machines," *Formal Methods Syst. Des.*, vol. 59, no. 1, pp. 77–102, 2021.

[34] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Uppaal — a tool suite for automatic verification of real-time systems," in *Hybrid Systems III*, R. Alur, T. A. Henzinger, and E. D. Sontag, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 232–243.

[35] R. M. Hierons, M. García Merayo, and M. Nuñez García, "Testing from a stochastic timed system with a fault model." *Journal of Logic and Algebraic Programming*, vol. 78, no. 2, pp. 98–115, January 2009.

[36] A. Tvardovskii, K. El-Fakih, and N. Yevtushenko, "Deriving tests with guaranteed fault coverage for finite state machines with timeouts," in *Testing Software and Systems*, I. Medina-Bulo, M. G. Merayo, and R. Hierons, Eds. Cham: Springer International Publishing, 2018, pp. 149–154.

[37] A. Tvardovskii, K. El-Fakih, , and N. Yevtushenko, "Testing and incremental conformance testing of timed state machines," *Science of Computer Programming*, vol. 233, 2024. [Online]. Available: https://doi.org/10.1016/j.scico.2023.103053

[38] 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

[39] 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

[40] U. C. Türker, R. M. Hierons, M. R. Mousavi, and I. Y. Tyukin, "Efficient state synchronisation in model-based testing through reinforcement learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 368–380.

[41] D. Neider, R. Smetsers, F. Vaandrager, and H. Kuppens, "Benchmarks for automata learning and conformance testing," Cham, pp. 390–416, 2019. [Online]. Available: https://automata.cs.ru.nl/Overview

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

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

[44] 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.

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

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

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

[48] 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.

[49] 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.

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

[51] A. Kisielewicz, J. Kowalski, and M. Szykuła, "A fast algorithm finding the shortest reset words," *Computing and Combinatorics*, vol. 7936, pp. 182–196, 2013.

[52] 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

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

[54] C. Güniçen, K. Inan, U. C. Türker, and H. Yenigün, "The relation between preset distinguishing sequences and synchronizing sequences," *Formal Asp. Comput.*, vol. 26, no. 6, pp. 1153–1167, 2014. [Online]. Available: http://dx.doi.org/10.1007/s00165-014-0297-8

[55] R. M. Hierons and U. C. Türker, "Parallel algorithms for generating distinguishing sequences for observable nondeterministic fsms," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 1, jul 2017. [Online]. Available: https://doi.org/10.1145/3051121

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

[57] 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

[58] 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.

[59] 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.

[60] 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

[61] 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.

[62] 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

[63] 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.

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

[65] 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.

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

[67] 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.

[68] 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

[69] 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.

[70] M. G. Lagoudakis, *Value Function Approximation*. Boston, MA: Springer US, 2010, pp. 1011–1021.

[71] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[72] 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

[73] U. C. Türker, T. Ünlüyurt, and H. Yenigün, "Effective algorithms for constructing minimum cost adaptive distinguishing sequences," *Information and Software Technology*, vol. 74, pp. 69–85, 2016.

[74] 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.

[75] 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.

[76] 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.

[77] F. Wilcoxon, *Individual Comparisons by Ranking Methods*. New York, NY: Springer New York, 1992, pp. 196–202.

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

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

[80] NVidia, "Producer of the GPU cards used in experiments." [Online]. Available: https://www.nvidia.com/en-gb/

[81] RFC1350, "The tftp protocol (revision 2)," in *URL: http://www.rfc-editor.org/rfc/rfc1350.txt*, 1992.
[82] RFC793, "Transmission control protocol," in *URL: https://datatracker.ietf.org/doc/html/rfc793*, 1992.
[83] D. Hogrefe, "Osi formal specification case study: the inres protocol and service," in *IAM-91-012 - Technical Report: University of Bern, Switzerland*, 1992.
[84] W. H. Chen, "Executable test sequences for the protocol data flow property," in *Formal Techniques for Networked and Distributed Systems*, 2011, pp. 285–299.
[85] A. Bhanpurawala, K. El-Fakih, and I. Zualkernan, "A formal assisted approach for modeling and testing security attacks in iot edge devicess," *arXiv:2210.05623*, 2022.
[86] B. Song, H. Choi, and H. S. Lee, "Surveillance tracking system using passive infrared motion sensors in wireless sensor network," in *2008 International Conference on Information Networking*, 2008, pp. 1–5.

Uraz Cengiz Türker is a Lecturer at the School of Computing and Communications at the University of Lancaster, Lancaster, U.K. He received the BSc, MSc, and PhD degrees in computer science and engineering from Faculty of Science and Egineering at the Sabanci University, Istanbul, Türkiye. Before joining the University of Lancaster (2022), he was a Lecturer at the School of Computing and Mathematical Sciences at the University of Leicester, Leicester, U.K.

Robert M. Hierons received the BA degree in mathematics from Trinity College, Cambridge, U.K., and the PhD degree in computer science from Brunel University, London, U.K. He joined the Department of Mathematical and Computing Sciences, Goldsmiths College, University of London, London, before returning to Brunel University in 2000. He was promoted to full professor in 2003 and joined The University of Sheffield, Sheffield, U.K., in 2018.

Khaled El-Fakih is a professor at the College of Engineering at the American University of Sharjah, which he joined in Sept. 2001. His research work includes formal testing, automatic synthesis of distributed systems, optimization, and application of metaheuristic algorithms.

Mohammad Reza Mousavi is a professor of Software Engineering at King's College London. Prior to that, he held positions at Leicester, Halmstad, Chalmers/Gothenburg, Reykjavik and Eindhoven. His areas of interest are model-based testing, particularly applied to testing variability-intensive, cyber-physical systems, quantum-classical hybrid systems and concurrency theory.

Ivan Y. Tyukin is a professor of Mathematical Data Science and Modelling at King's College London, U.K. Before joining King's, he worked at RIKEN Brain Science Institute, Japan, at the University of Leicester, U.K., and was an Adjunct Professor at the Norwegian University of Science and Technology (NTNU), Norway. His research interests span mathematical modelling, control, optimisation, data analysis, and mathematical foundations of AI and Machine Learning.