Quantitative Verification-Aided Machine Learning: A Tandem Approach for Architecting Self-Adaptive IoT Systems

Javier Cámara
University of York
York, United Kingdom
javier.camaramoreno@york.ac.uk

Henry Muccini
University of L'Aquila
L'Aquila, Italy
henry.muccini@univaq.it

Karthik Vaidhyanathan
Gran Sasso Science Institute
L'Aquila, Italy
karthik.vaidhyanathan@gssi.it

***Abstract*—Architecting IoT systems able to guarantee Quality of Service (QoS) levels can be a challenging task due to the inherent uncertainties (induced by changes in e.g., energy availability, network traffic) that they are subject to. Existing work has shown that machine learning (ML) techniques can be effectively used at run time for selecting self-adaptation patterns that can help maintain adequate QoS levels. However, this class of approach suffers from learning bias, which induces accuracy problems that might lead to sub-optimal (or even unfeasible) adaptations in some situations. To overcome this limitation, we propose an approach for proactive self-adaptation which combines ML and formal quantitative verification (probabilistic model checking).**

In our approach, ML is tasked with selecting the best adaptation pattern for a given scenario, and quantitative verification checks the feasibility of the adaptation decision, preventing the execution of unfeasible adaptations and providing feedback to the ML engine which helps to achieve faster convergence towards optimal decisions. The results of our evaluation show that our approach is able to produce better decisions than ML and quantitative verification used in isolation.

***Index Terms*—Self-Adaptive architectures, proactive adaptation, machine learning, reinforcement learning, Q-learning, quantitative verification, probabilistic model checking, IoT architectures, self-adaptation patterns, software architecture.**

I. Introduction

The advent of the Internet of Things (IoT) is transforming and shaping the way in which many tasks are supported across application domains. According to Gartner, 8.4 billion IoT devices were in use in 2017 and the projection is that this value will increase to 21 billion by 2020 [2]. Considering the tremendous increase in the amount of data traffic generated, as well as the energy consumed by these devices, reducing energy consumption and data traffic are considered as two of the most important QoS challenges in IoT [4], [5], [11]. In fact, energy efficiency is becoming a global concern, with ICT expected to consume around 21% of the world's electricity by 2030.

Despite their increasing adoption rate, architecting IoT systems still poses important challenges that relate to their inherently dynamic and heterogeneous nature, as well as to the uncertainties they are subject to, which are induced by different sources in their execution environment (e.g., fluctuations in resource availability, interactions between software and physical processes) [12]. Inability to mitigate the effects of these uncertainties can have major implications on the quality of service (QoS) levels offered by these systems [4], [5].

To improve this situation, recent years have seen the emergence of multiple architecture-based self-adaptation techniques aimed at maintaining and guaranteeing improved levels of QoS in applications deployed in different domains [8], [9]. In the specific area of IoT, architectural patterns for self-adaptation recently proposed [14] aim at maintaining acceptable QoS levels at run time. However, different patterns have disparate impacts and present different tradeoffs in QoS (e.g., performance, availability, energy consumption), depending on the operational context of the system at run time. This calls for *self-adaptive* techniques that can autonomously switch between patterns at run time, depending on the specific conditions of the execution context, to ensure that the system does not deviate from acceptable QoS levels.

To this end, machine learning (ML) techniques can be used to perform such adaptations because they can learn from QoS data, predict situations that might demand adaptation, and proactively select an adaptation pattern to mitigate the effects of potential QoS violations that might appear in the short term [48]. However, ML suffers from learning bias arising from data and algorithms, which can sometimes result in bad predictions, leading to sub-optimal, or even infeasible adaptations [15], [16], [17].

This is a problem where the use of Quantitative Verification (QV) techniques (and in particular, probabilistic model checking [54]) can improve on the current situation, *complementing* the strengths of ML: given a decision (architectural pattern selection), and a formal description of the current configuration of the system and its execution conditions, QV is able to provide feedback concerning the feasibility of the decisions selected by ML. This feedback, in the form of quantitative guarantees about expected QoS levels, is used to: (i) prevent the execution of infeasible solutions selected by ML, and (ii) improve the quality of future decisions made by ML.

Some existing approaches in self-adaptation propose using ML either as a means to predict the need for adaptation, for reducing the solution space, or to perform adaptation

selection [43]–[45], [48]. In contrast, we propose a novel way of performing proactive adaptation leveraging the use of ML and QV in such a way that, ML is responsible for making decisions and QV is used to *verify* those decisions, as well as to *provide feedback* to ML, helping it to achieve faster *convergence towards closer-to-optimal decisions*, opening up the possibility of finding better solutions than using either QV or ML in isolation.

The contribution in this paper is an approach for architecting self-adaptive IoT systems which uses the learning ability of ML, and the verification capabilities of QV to identify and enact optimal adaptations proactively. Given a scenario and a set of QoS (energy and data traffic) requirements, our approach (i) uses Reinforcement Learning (RL) to select an adaptation pattern, (ii) uses probabilistic model checking to verify the feasibility of the decisions made by RL with respect to the QoS requirements, (iii) executes the adaptation if the decisions are found to be feasible, and otherwise requests a new decision from RL, and (iv) provides feedback to RL for improving the quality of future decisions.

Evaluation results of our approach on the NdR case study (https://nottedeiricercatoriaq.it/) show remarkable increments in the satisfaction of QoS levels with respect to self-adaptation based on ML and QV used in isolation.

II. Motivating Scenario

We illustrate our approach on a scenario that concerns the NdR street fair, which is a scientific exhibition event organized by the University of L'Aquila. In this event, the research community and public (about 25,000 visitors) are brought together to participate in activities held in the city center, in which performances, lectures, demonstrations, and workshops take place in squares, main streets, and buildings.

Our research group has developed an IoT-based solution to improve the quality of the visiting experience. Without the loss of generality, in this paper we focus on two such IoT services planned for NdR. These services are related to the automated management of three venues (with capacity for 500 and 2×200 seats) and control of two automated parking lots (capacity 100 and 75 slots). These parking lots and venues are created in an ad-hoc way and on a temporary basis. Hence, availability of external power sources and legacy infrastructure is limited.

The system uses sensors at the venue and parking entrances and exits to gather real-time information on the venue and parking space availability. This information is then ingested by a central database and, once processed, results are sent to the display devices in the parking lots and venues to redirect users to alternative parking lots based on their venue preferences. Each of these sensors can operate in: (i) *normal mode*, in which the sensor component gathers data at standard rate (normal operational scenario, and (ii) *critical mode*, in which the sensor component gathers data with higher frequency (e.g., when there is a sudden increase in people/vehicle arrival/departure rate, the data needs to be gathered at a faster rate to ensure service accuracy).

The two main concerns in our scenario are to ensure that the system is efficient with respect to the energy and overall data traffic requirements. This is due to the fact that all sensors used are battery-powered. Even though charging can be performed from time to time, their charge might plummet during peak hours, and plugging in a new sensor or charging the existing one might be difficult while keeping the system available. Moreover, we also need to ensure that the overall data traffic level is not too low, to be able to maintain an acceptable service accuracy, nor too high, because network congestion might have a remarkable impact on system performance.

Hence, given: (i) data traffic constraints D_{max} and D_{min}, i.e., the maximum and minimum data traffic levels acceptable for a given execution period of duration τ, and (ii) E_{max} being the maximum energy that can be consumed for that same τ-period, the goal of the system is maximizing the following utility function that captures the non-functional requirements of our scenario and enables us to quantify their satisfaction:

$$U_\tau = w_e \cdot E_\tau + w_d \cdot T_\tau, \text{ with}$$

$$T_\tau = \begin{cases} (D_{max} - d_\tau) \cdot p_{dv} & \text{if } d_\tau \geq D_{max} \\ d_\tau - D_{min} & \text{if } D_{max} > d_\tau > D_{min} \\ (d_\tau - D_{min}) \cdot p_{dv} & \text{if } d_\tau \leq D_{min} \end{cases}$$

$$E_\tau = \begin{cases} E_{max} - e_\tau & \text{if } e_\tau < E_{max} \\ (E_{max} - e_\tau) \cdot p_{ev} & \text{otherwise} \end{cases}$$

where, e_τ, d_τ represent the total energy and data traffic consumed by the system for the τ-period, and w_e, $w_d \in \mathbb{R}^+$ are weights that capture the priority of energy and data traffic savings, respectively. T_τ and E_τ are piece-wise functions that capture the data traffic and energy savings respectively where, p_{ev}, $p_{dv} \in \mathbb{R}^+$ represent penalties for the violations of energy and data traffic thresholds, respectively.

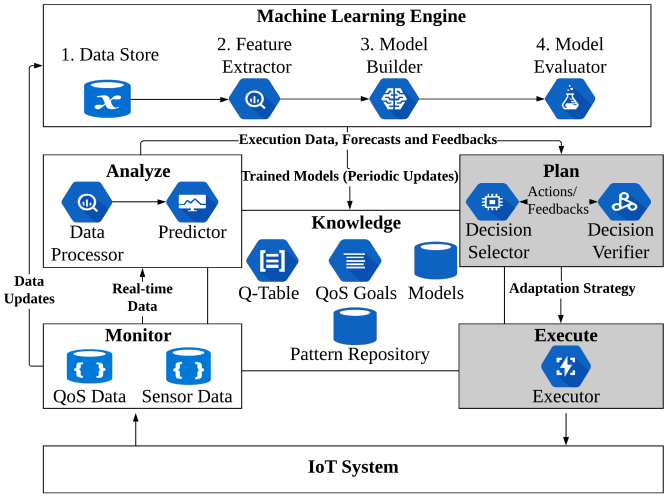

Fig. 1. Approach Overview

III. Approach Overview

The goal of our approach to architecting adaptive IoT systems is to ensure that QoS requirements are satisfied

throughout the execution of the system. To this end, the approach employs some of the main ideas behind model predictive control (MPC) [55], which include: (i) using models to forecast future system behaviour, (ii) the computation of a sequence of control (i.e., adaptation) actions, and (iii) using a receding horizon, i.e., repeating the selection of the sequence of control actions from the *actual system state* after the execution of the first control action. This latter part allows accounting for the effects of potential disturbances.

Structurally, the approach builds on the MAPE-K pattern [18], and instantiates its different components in the following way (Figure 1):

The *Monitor* activity regularly collects two types of data from the IoT system at run time, namely QoS metrics (e.g., instantaneous traffic, energy consumption) and the data collected by sensors. These are continually sent to the *Analyze* activity and the *Machine Learning Engine*.

The *Machine Learning Engine (MLE)* is responsible for building two types of ML models using Long Short-Term Memory (LSTM) Networks [13]: (i) QoS models for forecasting the expected energy consumption and data traffic up to the duration of the planning horizon, and (ii) forecast models predicting the expected behaviour of every sensor component up until the planning horizon. The MLE periodically updates these trained models in the *Knowledge Base*.

The *Analyze* activity is responsible for identifying the need for adaptation based on the data obtained. It first processes the data as required by the *Predictor* component using the *Data Processor* component. The *Predictor* component uses the processed data to predict the expected QoS of the system and behaviour of sensors (operational modes) using forecast models from the *Knowledge Base*. These forecasts are used to identify the need for adaptation. If adaptation is needed, the forecasts are made available to the *Plan* activity.

The *Plan* activity mainly consists of two components, (i) *Decision Selector* and (ii) *Decision Verifier*. The *Decision Selector* selects the best architectural pattern (from those available in the *Knowledge Base*) that can be used to perform an adaptation based on the forecasts received. It performs this selection using reinforcement learning. This selection is then fed to the *Decision Verifier* for further verification. This is to ensure that only decisions that are feasible with respect to QoS goals defined in the *Knowledge Base* are executed. In case of an infeasible decision, the decision verifier component sends negative feedback to the decision selector and requests a new decision. If the decision produced by the decision selector is feasible, then the decision verifier sends positive feedback to the decision selector and sends the decision to the *Execute* activity. The *Decision Selector* collects feedback from two sources for continuous improvement: (i) the verification results obtained as the immediate feedback for the decision made, and (ii) for every decision that was sent to the *Execute* activity, it uses the QoS forecast received in the next iteration of decision making as an additional source for feedback.

The *Execute* activity is responsible for enacting the selected adaptation obtained from the *Plan* activity via effectors embedded at the system level that enable architectural change in the IoT system.

The *Knowledge Base* acts as a central storage for different types of knowledge required by different layers for performing the adaptations. It stores four types of information: (i) *Q-Table*, which is a look up table used by the Decision Selector to select the best pattern for adaptation, (ii) *QoS goals*, which capture the acceptable energy and data traffic thresholds as defined by stakeholders such as hardware and network engineers, (iii) *Model Repository*, which contains the updated ML models for forecasting the QoS and expected operational modes of sensor components. These models are further used by the *Predictor* component of the *Analyze* activity, and (iv) *Pattern Repository*, which contains the set of adaptation patterns available for performing the adaptation and their definitions.

Our instantiation of the *Monitor* and *Analyze* activities, along with the MLE have been presented in previous work [23]. In this paper, we focus on the *Plan* and *Execute* activities (enclosed by grey boxes in Figure 1). In particular, we focus on how reinforcement learning (*Decision Selector*) and formal verification (*Decision Verifier*) via model checking can complement each other to guide the system towards selecting architectural adaptations that optimize the guarantees of satisfying acceptable QoS levels.

IV. Background

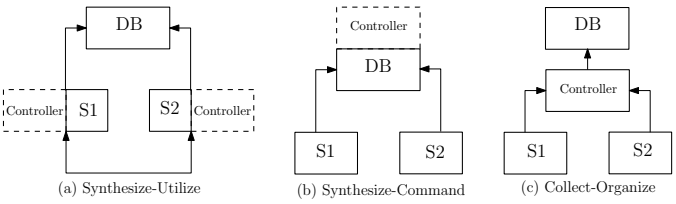

Fig. 2. Architectural Patterns for Self-Adaptation in IoT

A. Architectural Patterns for Self-Adaptation

Three patterns for self-adaptation in cyber-physical systems were identified by Musil et al. [14]. A simplified version of these patterns is illustrated in Figure 2:

a) Synthesize-Utilize (SU) is a fully decentralized pattern in which each of the sensor nodes has the ability to make decisions without resorting to an external controller.

b) Synthesize-Command (SC) is a centralized pattern in which all the sensor nodes and other components communicate with a central coordinator, which is in charge of making decisions.

c) Collect-Organize (CO) is a semi-decentralized pattern which uses additional controller components to receive the data from the sensor nodes and for making decisions, i.e., there will be multiple controllers acting as local coordinators for a group of sensor nodes.

B. Quantitative Verification

Quantitative Verification (QV), or *Probabilistic Model Checking* [54], is a set of formal verification techniques that enable modeling systems that exhibit stochastic behaviour, as

well as the analysis of quantitative properties that concern costs/rewards (e.g., resource usage, time) and probabilities (e.g., of invariant violation, reachability, etc.).

In QV, systems are modeled as state-transition systems augmented with probabilities such as discrete-time Markov chains (DTMC) and continuous-time Markov chains (CTMC), which is the formalism used in this paper to capture the behaviour of IoT systems with respect to their message exchange and energy consumption in a natural way:

Definition 1: A labelled Continuous-Time Markov Chain (CTMC) extended with rewards is a tuple $\mathcal{C} = (S, s_i, R, L, \iota)$, where S is a finite set of states, $s_i \in S$ is the initial state, $R : S \times S \to \mathbb{R}^+$ is the transition rate matrix, $L : S \to 2^{AP}$ is a labelling function which assigns to every state $s \in S$ a set $L(s)$ of atomic propositions valid in that state, and $\iota : S \times S \to \mathbb{R}^+$ is a transition reward function that assigns a reward every time a transition occurs in the CTMC.

In the definition above, the transition rate matrix R determines how transitions between states (e.g., capturing message exchanges between nodes in an IoT system) are triggered in a CTMC. Concretely, the probability of a transition being triggered within t time units is equal to $1 - e^{-R(s,s')\cdot t}$ (a transition of rate $1/t$ will take on average t time units to be triggered). Moreover, the reward assignment function can be used to encode rewards and costs, e.g., the energy consumed by devices every time a message is exchanged between two nodes in the network.

System properties are expressed using some form of probabilistic temporal logic, such as Continuous Stochastic Logic (CSL), which enables quantifying some probability or reward, or stating that they meet some threshold. In particular, CSL reward quantification properties can be employed to analyze the traffic and energy consumption in a IoT system described as a CTMC. For instance, the class of property $\mathsf{R}^{\mathsf{r}}_{=?}[\mathsf{C}^{<=\mathsf{t}}\ \phi]$ allows quantifying the reward r accrued up to time t across all execution paths of the system. An example of a property employing this operator for quantifying energy consumption in an IoT system might be $\mathsf{R}^{\mathsf{energy}}_{=?}[\mathsf{C}^{<=600}]$, meaning "accrued energy over the next 10 minutes (600 seconds) across all system execution paths."

V. Verification-Aided ML for Adaptive IoT

This section describes how the *Plan* activity uses reinforcement learning to select the best adaptation pattern and further, how it uses model checking as a guide to continually improve the decision making process using feedback obtained from verification. The decision making process, which involves six components, is shown in Figure 3. In the remainder of this section, we first introduce some preliminary definitions used in the approach, and then we explain each of the components and their interaction in detail.

A. Preliminaries

The overall goal of the approach is to ensure that the QoS requirements are satisfied throughout the execution of the system. These are formalized in QoS goals:

Definition 2 (QoS Goal): We define a QoS goal as a pair (μ, u), where $\mu \in M$ is a unique label identifier for a QoS metric, and $u = (u_l, u_h) \in \mathbb{R}^2$ is a pair of threshold values.

Example 1: In our case study, we define two QoS goals:

1. *Energy consumption*, $g_e = (energy, (e_l, e_h))$, states that the accrued energy consumed by the system should not exceed a maximum value e_h. In this goal, e_l represents the threshold below which the sensors save the maximum energy.

2. *Data traffic*, $g_d = (traffic, (d_l, d_h))$ states that the maximum total traffic allowed in the system should not exceed d_h, and should stay above d_l, which is the minimum traffic to be satisfied by the system to be able to maintain an acceptable service accuracy.

To achieve these QoS goals, our approach works by periodically generating an adaptation decision that considers predictions (forecasts) about the behaviour of the system over a time horizon of duration $H \in \mathbb{R}^+$. We assume a decision period of duration $\tau \in \mathbb{R}^+$, and consider H to be a multiple of τ. Hence, for a decision generated for time instant t: (i) the forecasts employed as model of the environment considered for the decision cover the period $[t, t + H]$, (ii) the decision made is executed (i.e., change of pattern, if the selected one is different from the current pattern active in the system), and (iii) the pattern selected is maintained as active for the time interval $[t, t + \tau]$. At that point $(t + \tau)$, a new decision is generated for the period $[t+\tau, t+2\cdot\tau]$ that considers forecasts for the period $[t + \tau, t + \tau + H]$. This sequence is continually repeated each τ-period.

Generating decisions employs two types of forecasts over the lookup horizon: (i) QoS forecasts and (ii) behaviour forecasts that capture the evolution of relevant system and environment variables.

Definition 3: A *QoS Forecast* a pair $(\mu, u) \in M \times \mathbb{R}^+$, where μ is a unique label identifier for a QoS metric, and u is its predicted value of the metric accrued over the duration of the lookup horizon.

Definition 4: A *Behaviour Forecast* is a sequence $\langle f_1, \ldots, f_k \rangle$ where each element $f_{i \in 1 \ldots k}$ is a tuple $(v, l, u) \in V \times \mathbb{R}^2 \times \mathcal{D}(v)$, where: v is a unique identifier for a system/environment variable, and l is a time interval during which v takes the value u. $\mathcal{D}(v)$ denotes the domain of v. We assume that the time intervals of the elements in f_i fully cover up to the duration of the lookup horizon H.

Example 2: Consider a horizon $H = 600$ seconds in our system. A sensor s may operate in a mode captured by variable $s.mode$, which takes values in the domain $\{normal, critical\}$. A behaviour forecast for the mode in which sensor is operating during the time interval $[t, t + 600]$ like $\langle (s.mode, [0, 100], normal), (s.mode, [101, 600], critical) \rangle$ captures that sensor s is expected to operate in normal mode for the next 100 seconds, and in critical mode during the remainder of the time, up until the end of the horizon.

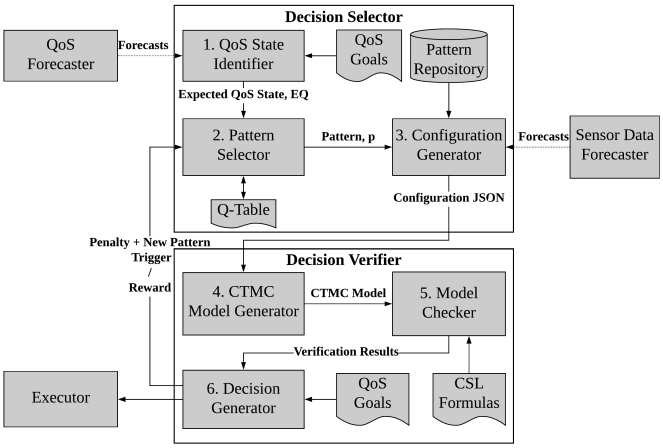

Fig. 3. Detailed View of Decision Making Process

B. QoS State Identifier

Our approach assumes a discrete set of QoS categories for each dimension of concern (e.g., energy consumption, traffic).[1] The first part of decision making is identifying the expected QoS state of the system EQ for the next τ-period, i.e., the set of expected QoS categories for every dimension of concern (Component 1, Figure 3).

For each decision period elapsed, the *QoS State Identifier* receives the set of QoS forecasts QF for the next decision period from the *QoS Forecaster* of the *Analyze* activity. Then, QF is mapped to a set of categories EQ that identifies the QoS state of the system for the next decision period. The mapping between QF and EQ is obtained based on the values in QF, and how they meet the thresholds stated in QoS goals.

For instance, let the QoS goal for energy consumption be $(energy, (0.5, 2))$, meaning that the energy consumption of the system up until the time horizon should stay in the range 0.5-2 Joules. We can define a mapping function $qc : \mathbb{R}^2 \rightarrow \{low, medium, high\}$ as $\{[0, 0.5] \mapsto low, [0.5, 2] \mapsto medium, [2, \infty] \mapsto high\}$. To identify the QoS state of the system EQ, this mapping process is repeated across all quality dimensions of concern for every element of QF. Once identified, EQ is provided as input to the *Pattern Selector* (Component 2, Figure 3).

C. Pattern Selector

The role of the *Pattern Selector* is to select the best adaptation pattern that can be applied based on the expected QoS state of the system EQ. It uses reinforcement learning [24], and in particular, Q-Learning [25], to decide on the adaptation pattern. The problem of selecting the best adaptation pattern can be converted into an instance of the well-known "Robot in the Grid World" problem [26], where the running software architecture can be considered as a robot that has to navigate through a grid where each location in the grid corresponds to a QoS state, and the goal is to keep moving to a position in the grid which enables the architecture to become optimal with respect to energy and data traffic goals as defined by the set of QoS goals.

The state space of our Q-Learning algorithm $S \subseteq 2^{QC} \times P$ is defined over the set of possible QoS states, and the set of available patterns $P = \{1 \dots m\}$. The algorithm also assumes a set of actions, $A = \{a_1, a_2..a_m\}$ that correspond to reconfigurations to patterns in P, and a reward function $\rho : S \rightarrow \mathbb{Z}$ that maps states to an integer reward assigned for moving into a state via a pattern change action.

The algorithm makes use of a lookup Q-Table matrix that can be encoded as a function $Q : S \times A \rightarrow \mathbb{R}$, that returns a real number (Q-Value) for any arbitrary state-action pair (s, a). This value gives an estimation of how valuable it is to select the action a from state s.

Pattern selection is presented in Algorithm 1. It takes as inputs the states, actions, expected QoS state and current pattern (S, A, EQ and p). Further inputs are reward function ρ, as well as parameters α and γ where, $0 \leq \alpha \leq 1$ represents the learning rate which captures the importance given to the learned observation at each step, and $0 \leq \gamma \leq 1$ represents the discount factor, which can be considered as the weight given to the next action.

Algorithm 1 Pattern Selection Algorithm

1: **procedure** PATTERN-SELECTOR(S,A,EQ,p,ρ,α,γ) ▷ States, Actions, QoS state, active pattern, learning rate and discount rate
2: $s^t \leftarrow (EQ, p)$
3: $a^t \leftarrow a_p$
4: $r^t \leftarrow \rho(s^t)$
5: $(s^{t+1}, a) \leftarrow argmax_{(s,a) \in S \times A} Q(s, a)$
6: $Q'(s^t, a^t) = (1-\alpha) \cdot Q(s^t, a^t) + \alpha \cdot (r^t + \gamma \cdot Q(s^{t+1}, a))$
7: **return** a

Before learning begins, Q is initialized to an arbitrary fixed value. The Expected QoS state, EQ along with the current execution pattern p, becomes the current state s^t and the action corresponding to the pattern p (denoted by a_p) is assigned to the current action a^t (lines 2 and 3). At each time t the agent selects an action a^t, observes a reward r^t, enters a new state s^{t+1}(that depends on the selected action – line 5), and Q is updated (line 6). The core of the algorithm is a simple value iteration update, using the weighted average of the old value and the new information (controlled by the α and γ parameters, respectively). The algorithm returns the action a, which corresponds to the selection of the new pattern. This action maximizes the Q-value that can be obtained by performing the different actions available for the next state s^{t+1} (line 5) ensuring that for every time instant, the best pattern for adaptation according to Q is selected.

The pattern selected by the algorithm is sent to the *Model checker* for verification through the *Configuration Generator*.

[1] We denote the set of all QoS categories across dimensions by QC.

D. Configuration Generator

The Configuration Generator is responsible for creating the configuration of the selected pattern at run time as required by the *CTMC Model Generator*. It uses the definitions of the patterns from the *Pattern Repository* along with the run-time execution data from the *Analyze* activity and the forecasts on the expected behaviour of sensors from the *Sensor Data Forecaster* to build the description of the different configurations.

The *Pattern Repository* contains the static information on the configuration of each adaptation pattern, which consists of a set of components involved C, and the set of connectors that exist between the components, $K \subseteq C \times C$.

Sensor components in C are annotated by properties such as the idle energy consumption and frequency of data transfers in different execution modes (for example, during critical situations such as emergency, sensors might communicate data more frequently, compared to a normal scenario).

Each connector $(c, c') \in K$ is annotated by properties such as the energy consumed for sending data from c to c', energy consumed by c for processing the data to be sent, and the energy consumed by c' for receiving the data from c.

Based on the pattern p selected by the *Pattern Selector*, the static configuration of the corresponding pattern is retrieved from the pattern repository and annotated with run-time information about sensor components, which includes the *Sensor Data Forecasts* obtained from the *Sensor Data Forecaster* component of the *Analyze* activity (details on how the forecasts are generated have already been presented in our previous work [23] and are left out of scope in this work). The forecast-annotated version of the configuration is then passed on to the CTMC Model Generator.

E. CTMC Model Generator

The CTMC model generator takes as input the configuration description obtained from the configuration generator, and produces a CTMC model analyzable via model checking. Concretely, the generator takes the specification of the set of components in the configuration, and for each one of them, it instantiates a process description using the different patterns shown in Figure 4. The mapping between component types in the class of IoT system we describe, and process types used in the CTMC models is shown in Table I.

TABLE I
COMPONENT-PROCESS TYPE INSTANTIATION FOR PATTERNS

Pattern / Comp.	Sensor	Database	Controller	Display
SU	Produce-Forward	Consume	- (in-sensor)	Consume
SC	Produce	Forward	Consume	Consume
CO	Produce	Consume	Forward	Consume

The *Produce/Produce-Forward* process type captures the behaviour of sensors, which can be in normal and critical mode (states N and C, respectively). Each one of the modes produces and sends sensor data at different rates (λ_n and λ_c) via action msg. Mode changes can be triggered by an exogenous event mode_chg (modeling, e.g., a variation in rate of cars entering a parking lot). In the SU pattern, this process type includes the additional states and transitions required to forward data received from other sensors (shown in dashed lines in Figure 4). In each one of the modes, the process can receive a message from any component it is connected to in the configuration (brackets denote multiple transitions), going into the states (N',C'), from which the message can be forwarded immediately (λ_i is a constant that denotes an instantaneous transition rate). Transitions that do not show a rate have a default value of 1, meaning that they are only triggered via synchronization on the events they are labeled with. Concerning reward structures, sending and receiving messages from node i accrues an energy cost of e_s^i and e_r^i units in reward structure energy, respectively, and 1 message for traffic in reward structure messages.

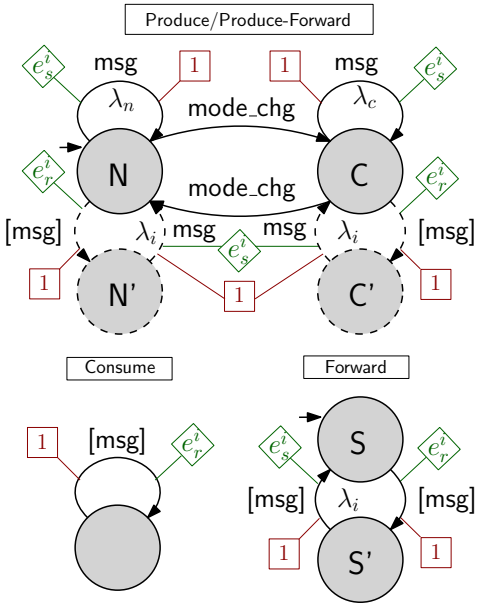

Fig. 4. CTMC process types for IoT nodes

The *Consume* process type is used to capture data sink nodes in the system (e.g., displays, the database in the CO and SU patterns). It just receives messages from any node in the network it is connected to, consuming energy e_r^i.

Finally, the *Forward* process type captures the behaviour of intermediate nodes in the network, which can receive messages from multiple nodes, and forward them to other nodes (e.g., controller in the CO pattern, database in the SC pattern).

The overall CTMC model results from the standard CSP parallel composition of all the processes instantiated, which synchronize on shared labels generated from the connections included in the configuration description.

F. Model Checker

The model checking component takes as input the CTMC model produced by the *CTMC model generator*, and is able to quantify the expected amount of energy consumed, as well the overall number of messages exchanged in the system for the

TABLE II
CSL PROPERTIES FOR MODEL CHECKING

Name	CSL Formula	Description
ϕ_e	$\mathsf{R}^{\mathsf{energy}}_{=?}[\mathsf{C}^{<=\mathsf{H}}]$	**Efficiency**: overall energy units consumed during H time units.
ϕ_t	$\mathsf{R}^{\mathsf{messages}}_{=?}[\mathsf{C}^{<=\mathsf{H}}]$	**Traffic**: overall number of messages exchanged during H time units.

time frame of the lookup horizon H. Quantification of each one of these properties is achieved via model checking of the CSL properties shown in Table II, for which the component uses PRISM's model checking engine.

G. Decision Generator

The *Decision Generator* is responsible for deciding the feasibility of the decision produced by the *Pattern Selector* based on the results of the analysis obtained from the *Model Checker*. The output from the model checker consists of the expected energy consumption and data traffic of the system while using the selected pattern. The decision generator verifies if these values are within the thresholds specified in *QoS goals*. If the expected energy consumption is less than the threshold, e_h as defined in QoS goal g_e and the expected data traffic is within the thresholds as defined in QoS goal g_d, then the decision is considered as a feasible (valid) one, and a positive feedback in the form of a reward, $r_m > 0$ is fed sent to the *Pattern Selector*. Otherwise, the decision is considered as infeasible (invalid), and a negative feedback in the form of penalty $p_m < 0$ is sent back to the *Pattern Selector*, along with the request of a different pattern.

Hence, for every decision made by the *Pattern Selector*, a verification step is performed by the *Model Checker*, improving the chances that only adaptations that are feasible are executed. Given an adaptation scenario, this combination provides multiple advantages: (i) it helps the system to select decisions based on past feedback obtained from the model checker as well as from the previous forecasts, (ii) it ensures that, even if RL generates a bad decision, it does not affect the system execution due to the involvement of the model checker, and (iii) it ensures that the model checker does not have to perform exploration of a broader solution space, because it just has to analyze system behaviour under the selected pattern.

VI. Implementation Overview

We implemented our approach using a traditional layered architecture with an enterprise-grade big data stack. We used a customized version of CupCarbon [27], [28] for realizing the architecture of the NdR case study and its corresponding simulations. Concretely, we modified CupCarbon to support dynamic pattern changes based on the decisions produced.

For generating the forecast models, we employed Python, along with the deep learning library Keras [34]. The *Pattern Selector* implementing Q-Learning was written in Python. The *Model Checker* component is written in Java and makes use of the PRISM model checker [56] API. The integration between the *Decision Maker* and the *Model Checker* was done using JPype [32]. Additionally, a web service implemented in Python using the Tornado framework [33] is used for communicating the pattern change to CupCarbon. The communication between CupCarbon and the different MAPE-K activities is powered by Apache Kafka [31].

VII. Experimentation and Results

The objective of our evaluation is to assess the effectiveness and efficiency of the approach by answering:

RQ1. How effective is the approach with respect to satisfying overall energy and data traffic goals?
RQ2. How much does using model checking along with ML improve satisfaction of goals over the use of just ML?
RQ3. What is the efficiency of adaptations?
RQ4. What is the computation overhead of adaptation?

In the remainder of this section, we first describe our experimental setup, as well as the data and metrics used for the evaluation of the approach, following with a discussion of the evaluation questions informed by our results.

A. Experimental Setup

Our testbed was deployed on two VM instances in Google Cloud. The first one run on a N1-Standard-4 CPU Intel Skylake Processor comprising 4 vCPU with 16 GB RAM. This instance was used for running the CupCarbon simulation and the producers for sending the QoS metrics and Sensor data to the Kafka broker. The second one runs on a N1-Standard-8 CPU with Intel Skylake Processor comprising 8 vCPU with 32 GB RAM. This was used for running the Kafka broker and the MAPE activities of our approach.

To simulate the real scenario of the case study with as much fidelity as possible, we created a script that generates data for each of the sensor components using intervals of 60 seconds with arrival rates based on a Poisson distribution for a period of 24 hours. The mean values of the distribution were selected based on observations from real NdR scenarios.

We evaluated the approach by performing the simulation of the case study using six different approaches for a period of 24 hours. Three of the approaches consisted of simulating the case study, fixing each of the patterns (SU, SC and CO). The other three approaches are as follows:

1. RL: Adaptation using just reinforcement learning (Q-Learning) as described in Section V.
2. MC: Approach which performs adaptation just based on model checking, but without using Q-Learning. For every decision period, it performs model checking to identify the best pattern by finding which pattern gives the maximum benefit for the thresholds specified.
3. RLMC: Our approach, which performs adaptation combining Q-Learning and model checking.

Considering the operational constraints we have from the case study, we defined the set of QoS goals, $QG = \{(energy, (10.0, 6.0)), (traffic, (2500, 3000))\}$, with energy

TABLE III
ENERGY CONSUMPTION AND DATA TRAFFIC COMPARISON (#DV : NUMBER OF DATA TRAFFIC VIOLATIONS, #EV : NUMBER OF ENERGY VIOLATIONS)

Approach	Energy (Joules)	Data Traffic (# messages)	Utility Score	# Max DV	# Min DV	# EV
CO	752.43	437885	186.89	90	0	0
SU	1995.85	382896	61.11	5	20	143
SC	1498.57	398393	158.11	14	8	104
RLMC	851.35	400239	355.24	17	6	2
MC	1246.43	407107	218.43	26	3	25
RL	1051.38	398294	272.98	18	7	8

measured in Joules and traffic in # of messages exchanged. We consider a horizon $H = 600$ seconds and $\tau = 60$ seconds. We use a learning rate $\alpha = 0.02$ and discount factor $\gamma = 0.2$ for performing the Q-Learning. The complete implementation along with the source code, datasets and ML models used for forecasts can be found here.[2]

B. Evaluation Metrics

To measure the effectiveness of the approach, we introduce four evaluation metrics: (i) Max and Min DV, which capture the number of violations of maximum (d_h) and minimum (d_l) data traffic thresholds as defined in g_d (c.f. Example I), (ii) EV, capturing the number of violations of the maximum energy threshold (e_h) as defined in g_e, and (iii) Utility Score (U) as defined in Section II, using normalized values for energy consumed e_τ and data traffic d_τ for every τ-period. We set $w_e = 2$, $w_d = 5$, $p_{ev} = 0.3$, and $p_{dv} = 0.5$. We assign a slightly higher weight to the traffic term (and also higher value to its penalty) because, although saving energy is a priority, we do not want to do it at the expense of a system that does not operate with the required accuracy.

C. Results

RQ1. *How effective is the approach with respect to satisfying overall energy and data traffic goals?*

To measure effectiveness, we calculate the total energy and data traffic consumed during simulation by each of the approaches. The aggregated results for our evaluation metrics are reported in Table III. The table shows that CO consumes the least energy, but at the same time, it maximizes data traffic. This is due to the semi-decentralized nature of the pattern which results in an increased amount of exchanged messages, resulting from the presence of extra controller components in the architecture. SU is the pattern that presents the lowest traffic volume, although to a level that is detrimental to maintain service accuracy (presents more than double Min DV count, compared to other approaches). SU is also the least energy-efficient. In contrast, SC is more energy-efficient than SU, but presents a higher data traffic volume with a high count of Max DV. This is due to the fact that every decision has to be taken by the centralized controller and hence the sensors need to send information to the database at a faster rate compared to SU, where sensors are equipped with decision making abilities. MC consumes less energy than SU and SC,

[2]https://github.com/karthikv1392/IoT_RLMC/

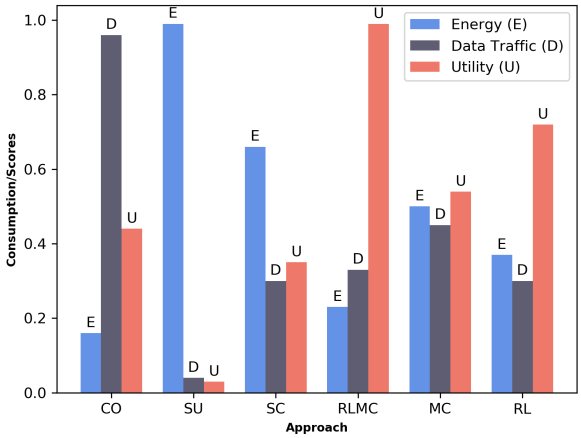

Fig. 5. A bar plot of the different approaches and their respective normalized energy and traffic consumption along with overall utility scores

also with lower traffic than CO. It offers better utility compared to fixed patterns because it can choose what is determined to be more adequate for different decision episodes. However, RL offers much better Utility and consumes less energy with lower traffic compared to MC. This can be attributed to the ability of RL to learn from feedback over time. Furthermore, RLMC is the most energy-efficient, compared to RL and MC and only at a slightly higher traffic volume than RL. This yields an increment in utility of 39% and 63% with respect to RL and MC, respectively. This clearly shows the remarkable impact that QV has on RL for decision making. The normalized values of energy, traffic and utility are shown in 5. RLMC scores the highest utility, being second to CO in energy efficiency. Although RL has lower data traffic than RLMC, the ratio of energy saved/traffic saved for RLMC (0.78/0.62 = 1.26) is higher than that of RL (0.62/0.75 = 0.83).

RQ2. *How much does using model checking along with ML improve satisfaction of goals over the use of just ML?*

To answer this question, we compare the cumulative utility score of all approaches (Figure 6). The plot shows how accrued utility starts diverging marginally during the initial stage, but then the gap between RL/RLMC and other approaches keeps on increasing. MC still offers better performance compared to each of the fixed patterns because it selects what it expects to give the best utility at the start of every decision period.

However, it does not have a way to improve the decision in the next iteration based on the feedback of the past decision. In contrast, both RL and RLMC have the advantage of feedback which allows them to progressively improve their decisions. Unlike in RLMC, the feedback in RL is obtained only after execution of the selected adaptation, which might end up being sub-optimal. The effect of these sub-optimal decisions can be clearly observed in the graph, where RLMC offers initially an increment of just 1% in utility over RL, but as time progresses, this value increases up to 39% over a span of 24 hours.

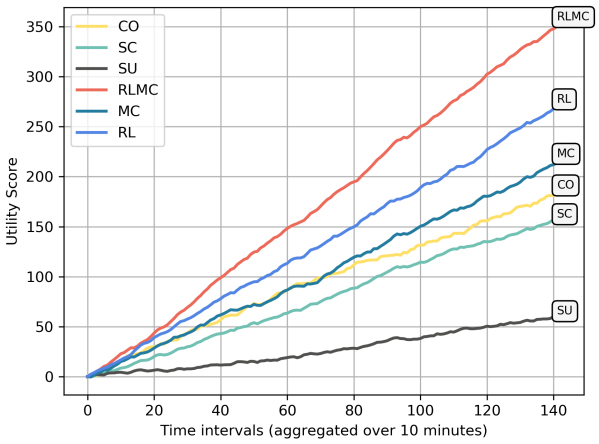

Fig. 6. Cumulative Utility scores for each of the approaches

RQ3. *What is the efficiency of adaptations?*

We evaluate adaptation efficiency in terms of the number of corrections made by the model checker during each decision period. A scatter plot showing the number of corrections made per interval can be seen in Figure 7. The figure shows that in between intervals, there is an increase in the number of corrections, which goes as high as 6. At the start of simulation, correction count is high for the first two adaptation cycles (due to the time required for initial learning). Then, we can observe that the number of corrections made remains 0 until 160 minutes. This again increases and RLMC is able to continue without many corrections for some time until the next peak. The next peak is given when MC identifies that a decision at a given point is not feasible based on the expected context (behaviour/mode changes of sensors).

This behaviour illustrates the ability of RLMC to learn and improve from the results obtained by the model checker, as well as from the feedback obtained from ML forecasts. On average, the number of corrections amounts to 1, but there are instances where it is 0 and few instances where it is as high as 6 where RL is forced to generate decisions, incurring high penalties (negative rewards) and additional computation overhead. It is due to this effect that even with such a high increment in the number of corrections, RLMC is still able to continue operating for some time without need for new corrections. This indicates that RLMC is very efficient with respect to the number of corrections performed by the model checker.

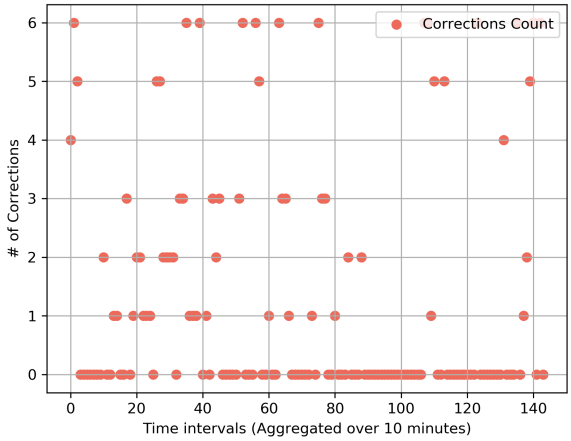

Fig. 7. Number of corrections performed by MC per adaptation

RQ4. *What is the computation overhead of adaptation?*

To evaluate adaptation efficiency in terms of computation overhead, we clocked the time required to generate adaptation decisions. The results show that on average, RLMC takes approximately 2 seconds for generating an adaptation decision. From that time, the fraction employed by RL amounts to approximately 0.23 seconds because it consists of simple look up operations and state update. Each correction operation from model checker takes close to 1 second. Despite the overhead introduced by the model checker, the observed decision times are reasonable in the context of the class of IoT application described, showing feasibility of RLMC.

VIII. Threats to Validity

Threats to *construct validity* concerns the decisions made due to incorrect forecasts which could arise from the *Analyze activity*. It might happen that the model forecasts a high energy consumption whereas the system would not have entered such a state and this leads to lower utility score. We understand this issue and since the same prediction model and sensor data sets are used for the evaluation for all the three approaches, this does not over-weights or under-weights the overall utility score of any approach.

Threats to *external validity* concerns the generalizability and scalability of our approach. Although our approach has been applied on a specific case study, it uses techniques that can be generalized to other classes of IoT system with similar concerns (energy consumption, performance, availability). Moreover, we believe that our approach can be applied to more complex systems with larger number of components with optimization of reinforcement learning and model checking components (e.g., using statistical model checking to improve scalability). In addition, the layered architecture used for implementation supports both horizontal and vertical scalability.

IX. Related Work

Self-adaptation in IoT/CPS: Weyns et al. in [35] propose an approach for managing run-time uncertainities associated with over-provisioning of resources in IoT systems. Fahad et al. [36] propose the notion of *emergent configuration* for engineering self-adaptive goal-oriented IoT systems. Model Driven Engineering (MDE) based approaches [37]–[39] use the concept of *models@run.time* and MAPE-K loops for adaptation. Nascimento et al. [40] propose an agent-based framework for performing self-adaptation for IoT applications. These approaches perform adaptations reactively and they do not acquire knowledge from the performed adaptations.

ML in self-adaptation: Some works that use ML to support self-adaptation in other domains include planning and adaptation in software systems [44], [45] where RL is used to identify the best adaptation strategy in a reactive manner and without any guarantees about the correctness of decisions. Anaya et al. [46] describe an approach for self-adaptive sensor networks that uses ML to predict the need for adaptation based on a forecast horizon. Idziak et al. [47] present an analysis of different decision-making techniques in self-adaptive systems, where reactive adaptations were carried out for a simple VM replacement problem using different learning algorithms. Chen et al. [48] present a multi-learner approach for self-adaptive and online QoS modeling for cloud-based software services where ML is used to predict the need for adaptation. A self-adaptive mechanism based on reinforcement learning for adaptation in autonomous systems was proposed by Mallozzi et al. [49], where RL is used reactively.

ML and QV in self-adaptation: There have been some works that use ML and QV for performing self-adaptation. Jamshidi et al. [41] describe an approach that uses ML to find a set of Pareto-optimal configurations in a large configuration space which are then used to identify the best adaptation plan using probabilistic model checking in autonomous mobile robots. Van Der Donckt et al. [42] also employ ML to identify a subset of adaptation options from a larger adaptation space and further select the best option using using cost-benefit based analysis. An approach that enhances traditional MAPE-K loop to support the use of ML for efficiently analyzing large adaptation spaces was presented by Quin et al. [43] where ML was shown to be effective in analyzing large adaptation space to identify the best solution candidates and further select the best decision using statistical model checking. However, these are reactive approaches where ML is used to aid QV and not the other way around. Also ML does not receive feedback to improve its decision making based on results of QV.

There has also been some work done in proactive adaptation. Moreno et al. employ time series forecasts combined with probabilistic model checking [50] and stochastic dynamic programming [51], [52] to perform proactive latency-aware adaptations based on a look-ahead horizon considering the uncertainty that may arise from the environment. Another approach, also based on principles of model predictive control with analytical models that capture the relation between control parameters and system outputs was proposed by Angelopoulos et al. [53]. In their approach, models are used to predict system behaviour and compose adaptation plans. While Moreno's approach [50]–[52] does not make use of feedback from performed adaptations to continually improve decision making, Angelopoulos' [53] does, although it focuses on parameter value tuning as control actions instead of the complex structural changes present in (IoT) architectures.

Our approach uses a combination of ML and QV for *proactive decision making* and, unlike all the existing approaches mentioned above, it employs QV as a *means to verify the feasibility of decisions produced by ML* with respect to the system context. It further uses the feedback of the verification to help ML achieve faster convergence towards closer-to-optimal decisions. Works that combine ML and QV employ ML as an aid for reducing the decision space for QV.

X. Conclusions and Future Work

This paper shows how ML and QV can be combined to architect self-adaptive IoT systems able to optimize the guarantees of satisfying acceptable QoS levels with respect to energy consumption and network traffic. Our evaluation shows that the approach exhibits a remarkable improvement (39% and 63%) over the use of ML and QV in isolation. Although the use of QV adds computational overhead with respect to using just ML, our results also show feasibility of the approach, which is able to produce decisions within a reasonable timescale for IoT applications.

With respect to future work, a first research avenue that we plan to explore is scalability. This will entail considering more QoS parameters and patterns, extending and validating the approach for performing adaptation by identifying more patterns and factoring into decision making additional QoS parameters like response time, utilization, and throughput, as well as their trade offs. A second extension of our approach concerns considering finer-grained topological changes, instead of the more general, coarse-grained patterns that we are currently using. This will enable us to explore a much richer solution space, in which systems exhibit hybrid topologies with respect to the patterns considered (e.g., a part of the system may be configured as Synthesize-Utilize, whereas another part operates in a Synthesize-Command style), potentially leading to closer-to-optimal solutions with respect to the existing approach. Finally, our approach currently assumes that forecasts can be produced with acceptable accuracy. We plan on assessing the robustness of our approach with respect to uncertainties that affect the accuracy of the forecasts employed for decision making, studying how the quality of the decisions degrade with higher levels of uncertainty, and devising extensions to mitigate the degradation of decision quality.

Acknowledgment

This material is based on work supported by Google Cloud.

REFERENCES

[1] A. Taivalsaari and T. Mikkonen. A roadmap to the programmable world: Software challenges in the iot era. IEEE Software, 34(1):72–80, Jan 2017. ISSN 0740-7459.

[2] Gartner, (2017).[online] Available at: https://tinyurl.com/hd8b9l8 [Accessed 5 Feb. 2020].

[3] Metzger, Andreas, Osama Sammodi, and Klaus Pohl. "Accurate proactive adaptation of service-oriented systems." In Assurances for Self-Adaptive Systems, pp. 240-265. Springer, Berlin, Heidelberg, 2013.

[4] Hammoudi, Sarra, Zibouda Aliouat, and Saad Harous. "Challenges and research directions for Internet of Things." Telecommunication Systems 67, no. 2 (2018): 367-385.

[5] Hassan, Zozo & Ali, Hesham & M Badawy, Mahmoud. (2015). Internet of Things (IoT): Definitions, Challenges, and Recent Research Directions. International Journal of Computer Applications. 128. 975-8887.

[6] Gubbi, Jayavardhana, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. "Internet of Things (IoT): A vision, architectural elements, and future directions." Future generation computer systems 29, no. 7 (2013): 1645-1660.

[7] Alreshidi, Abdulrahman, and Aakash Ahmad. "Architecting Software for the Internet of Thing Based Systems." Future Internet 11, no. 7 (2019): 153.

[8] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst., 4(2):14:1-14:42, May 2009. ISSN 1556-4665.

[9] Danny Weyns. Software engineering of self-adaptive systems: an organised tour and future challenges. Chapter in Handbook of Software Engineering, 2017

[10] Krupitzer, Christian, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. "A survey on engineering approaches for self-adaptive systems." Pervasive and Mobile Computing 17 (2015): 184-206.

[11] Atzori, Luigi, Antonio Iera, and Giacomo Morabito. "The internet of things: A survey." Computer networks 54, no. 15 (2010): 2787-2805.

[12] Esfahani, Naeem, and Sam Malek. "Uncertainty in self-adaptive software systems." In Software Engineering for Self-Adaptive Systems II, pp. 214-238. Springer, Berlin, Heidelberg, 2013.

[13] Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9, no. 8 (1997): 1735-1780.

[14] Musil, Angelika, Juergen Musil, Danny Weyns, Tomas Bures, Henry Muccini, and Mohammad Sharaf. "Patterns for self-adaptation in cyber-physical systems." In Multi-disciplinary engineering for cyber-physical production systems, pp. 331-368. Springer, Cham, 2017.

[15] Nature.com. (2020). Bias detectives: the researchers striving to make algorithms fair. [online] Available at: https://www.nature.com/articles/d41586-018-05469-3 [Accessed 5 Feb. 2020].

[16] Mehrabi, Ninareh, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. "A survey on bias and fairness in machine learning." arXiv preprint arXiv:1908.09635 (2019).

[17] Jabbari, Shahin, Matthew Joseph, Michael Kearns, Jamie Morgenstern, and Aaron Roth. "Fairness in reinforcement learning." In Proceedings of the 34th International Conference on Machine Learning-Volume 70, pp. 1617-1626. JMLR. org, 2017.

[18] Kephart, J. O., Chess, D. M. (2003). The vision of autonomic computing. Computer, (1), 41-50.

[19] Warrier, Maya M., and Ajay Kumar. "Energy efficient routing in Wireless Sensor Networks: A survey." In 2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET), pp. 1987-1992. IEEE, 2016.

[20] Andrae, Anders & Edler, Tomas. (2015). On Global Electricity Usage of Communication Technology: Trends to 2030. Challenges. 6. 117-157. 10.3390/challe6010117.

[21] Malmodin, Jens, and Dag Lundén. "The energy and carbon footprint of the global ICT and E&M sectors 2010–2015." Sustainability 10, no. 9 (2018): 3027.

[22] Stack, T. (2020). Internet of Things (IoT) Data Continues to Explode Exponentially. Who Is Using That Data and How? - Cisco Blogs. [online] Cisco Blogs. Available at: http://tinyurl.com/y6bf2cgj [Accessed 5 Feb. 2020].

[23] Henry Muccini and Karthik Vaidhyanathan, "Leveraging Machine Learning Techniques for Autonomous Decision Making in Adaptive Architectures", DISIM, University of L'Aquila, L'Aquila, Italy, TRCS: 001/2019, Jul. 16, 2019. Accessed on: Feb. 14, 2020. [Online]. Available: https://tinyurl.com/y445f45k

[24] Sutton, Richard S., and Andrew G. Barto. Introduction to reinforcement learning. Vol. 135. Cambridge: MIT press, 1998.

[25] Watkins, Christopher JCH, and Peter Dayan. "Q-learning." Machine learning 8, no. 3-4 (1992): 279-292.

[26] Tokic, Michel, and Haitham Bou Ammar. "Teaching reinforcement learning using a physical robot." In Proceedings of the ICML Workshop on Teaching Machine Learning. 2012.

[27] Bounceur, Ahcène. "CupCarbon: a new platform for designing and simulating smart-city and IoT wireless sensor networks (SCI-WSN)." In Proceedings of the International Conference on Internet of things and Cloud Computing, p. 1. ACM, 2016

[28] Cupcarbon.com. (2020). CupCarbon - A Smart City IoT WSN Simulator. [online] Available at: http://www.cupcarbon.com [Accessed 5 Feb. 2020].

[29] Chernyshev, Maxim, Zubair Baig, Oladayo Bello, and Sherali Zeadally. "Internet of Things (IoT): research, simulators, and testbeds." IEEE Internet of Things Journal 5, no. 3 (2018): 1637-1647.

[30] Lopez-Pavon, Cristina & Sendra, Sandra & F. Valenzuela-Valdes, Juan. (2018). Evaluation of CupCarbon Network Simulator for Wireless Sensor Networks. Network Protocols and Algorithms. 10. 10.5296/npa.v10i2.13201.

[31] Kreps, Jay, Neha Narkhede, and Jun Rao. "Kafka: A distributed messaging system for log processing." In Proceedings of the NetDB, pp. 1-7. 2011.

[32] Jpype.readthedocs.io. (2020). JPype documentation — JPype 0.7.1 documentation. [online] Available at: https://jpype.readthedocs.io/en/latest/ [Accessed 5 Feb. 2020].

[33] Tornadoweb.org. (2020). Tornado Web Server — Tornado 6.0.3 documentation. [online] Available at: https://www.tornadoweb.org/en/stable/ [Accessed 5 Feb. 2020].

[34] Keras.io. (2020). Home - Keras Documentation. [online] Available at: https://keras.io [Accessed 5 Feb. 2020].

[35] Weyns, Danny, Gowri Sankar Ramachandran, and Ritesh Kumar Singh. "Self-managing internet of things." In International Conference on Current Trends in Theory and Practice of Informatics, pp. 67-84. Edizioni della Normale, Cham, 2018.

[36] Alkhabbas, Fahed & Spalazzese, Romina & Davidsson, Paul. (2018). ECo-IoT: An Architectural Approach for Realizing Emergent Configurations in the Internet of Things. 86-102.

[37] Mirko D'Angelo, Annalisa Napolitano, and Mauro Caporuscio. Cyphef: a model drivenp engineering framework for self-adaptive cyber-physical systems. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, pages 101-104. ACM, 2018.

[38] Acosta Padilla, Francisco Javier. "Self-adaptation for Internet of things applications." PhD diss., Rennes 1, 2016.

[39] Federico Ciccozzi and Romina Spalazzese. Mde4iot: supporting the internet of things with model-driven engineering. In International Symposium on Intelligent and Distributed Computing, pages 67-76. Springer, 2016.

[40] do Nascimento Nathalia Moraes, and Carlos José Pereira de Lucena. "FIoT: An agent-based framework for self-adaptive and self-organizing applications based on the Internet of Things." Information Sciences 378 (2017): 161-176.

[41] Pooyan Jamshidi, Javier Cámara, Bradley Schmerl, Christian Kästner, and David Garlan. 2019. Machine learning meets quantitative planning: enabling self-adaptation in autonomous robots. In Proceedings of the 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '19). IEEE Press, Piscataway, NJ, USA, 39-50.

[42] Van Der Donckt, J., Weyns, D., Iftikhar, M. U., & Buttar, S. S. (2018, March). Effective Decision Making in Self-adaptive Systems Using Cost-Benefit Analysis at Runtime and Online Learning of Adaptation Spaces. In International Conference on Evaluation of Novel Approaches to Software Engineering (pp. 373-403). Springer, Cham.

[43] Quin, Federico, Thomas Bamelis, Singh Buttar Sarpreet, and Sam Michiels. "Efficient analysis of large adaptation spaces in self-adaptive systems using machine learning." In 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 1-12. IEEE, 2019.

[44] Kim, Dongsun, and Sooyong Park. "Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software." In Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS'09. ICSE Workshop on, pp. 76-85. IEEE, 2009.

[45] Han Nguyen Ho and Eunseok Lee. Model-based reinforcement learning approach for planning in self-adaptive software system. In Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication, page 103. ACM, 2015.

[46] Anaya, Ivan Dario Paez, Viliam Simko, Johann Bourcier, Noël Plouzeau, and Jean-Marc Jézéquel. "A prediction-driven adaptation approach for self-adaptive sensor networks." In Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 145-154. ACM, 2014.

[47] Pawel Idziak and Siobhan Clarke. An analysis of decision-making techniques in dynamic, self-adaptive systems. In Self-Adaptive and Self-Organizing Systems Work- shops (SASOW), 2014 IEEE Eighth International Conference on, pages 137-143. IEEE, 2014.

[48] Tao Chen and Rami Bahsoon. Self-adaptive and online qos modeling for cloud-based software services. IEEE Transactions on Software Engineering, 43(5):453-475, 2017.

[49] Piergiuseppe Mallozzi. Combining machine-learning with invariants assurance techniques for autonomous systems. In Proceedings of the 39th International Conference on Software Engineering Companion, pages 485-486. IEEE Press, 2017.

[50] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2015. Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015). ACM, New York, NY, USA, 1-12.

[51] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2016. Efficient Decision-Making under Uncertainty for Proactive Self-Adaptation. In Proceedings of the 2016 IEEE International Conference on Autonomic Computing (ICAC 2016). IEEE Computer Society. 147-156. 2016.

[52] Gabriel A. Moreno, Javier Cámara, David Garlan, Bradley R. Schmerl. Flexible and Efficient Decision-Making for Proactive Latency-Aware Self-Adaptation. ACM Transactions on Autonomous and Adaptive Systems 13(1): 3:1-3:36. ACM. 2018.

[53] Konstantinos Angelopoulos, Alessandro V. Papadopoulos, Vítor E. Silva Souza, and John Mylopoulos. 2016. Model predictive control for software systems with CobRA. In Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '16). ACM, New York, NY, USA, 35-46.

[54] Kwiatkowska, M.Z., Norman, G., Parker, D.: Stochastic model checking. In: FM for Performance Evaluation, 7th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems. LNCS, vol. 4486. Springer (2007)

[55] E. Camacho and C. Bordons, *Model Predictive Control*, ser. Advanced Textbooks in Control and Signal Processing. Springer London, 2004.

[56] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification*, volume 6806 of *LNCS*. Springer, 2011.