2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)

Efficient Analysis of Large Adaptation Spaces in Self-Adaptive Systems using Machine Learning

[1]Federico Quin, [1,2]Danny Weyns, [1]Thomas Bamelis, [2]Sarpreet Singh Buttar, [1]Sam Michiels
[1]*Katholieke Universiteit Leuven, Belgium*
[2]*Linnaeus University, Vaxjo, Sweden*
federico.quin@student.kuleuven.be, danny.weyns@kuleuven.be, sb223ce@student.lnu.se

***Abstract*—When a self-adaptive system detects that its adaptation goals may be compromised, it needs to determine how to adapt to ensure its goals. To that end, the system can analyze the possible options for adaptation, i.e., the adaptation space, and pick the best option that achieves the goals. Such analysis can be resource and time consuming, in particular when rigorous analysis methods are applied. Hence, exhaustively analyzing all options may be infeasible for systems with large adaptation spaces. This problem is further complicated as the adaptation options typically include uncertainty parameters that can only be resolved at runtime. In this paper, we present a machine learning approach to tackle this problem. This approach enhances the traditional MAPE-K feedback loop with a learning module that selects subsets of adaptation options from a large adaptation space to support the analyzer with performing efficient analysis. We instantiate the approach for two concrete learning techniques, classification and regression, and evaluate the approaches for two instances of an Internet of Things application for smart environment monitoring with different sizes of adaptation spaces. The evaluation shows that both learning approaches reduce the adaptation space significantly without noticeable effect on realizing the adaptation goals.**

***Index Terms*—self-adaptation, analysis, learning, IoT**

I. Introduction

Modern software systems are expected to deal with uncertain operating conditions; e.g., changing network conditions and dynamic traffic in Internet of Things (IoT). Without proper mitigation of these uncertainties, system qualities such as reliability and energy efficiency may be jeopardized [1]–[3]. In our research, we study how self-adaptation can be applied to provide guarantees for the qualities of software systems that operate under uncertainties. In this paper, we focus on uncertainties in the environment (execution context) [3] that can be categorized as "known unknowns". We refer to the qualities that are subject of self-adaptation as the *adaptation goals*. Aligned with other researchers in this area, we apply formal techniques at runtime to support the decision making for adaptation and provide guarantees for the adaptation goals, see e.g., [4]–[11].

Self-adaptation is typically realized by means of an external feedback loop system that is added to the software system [12]–[15]. When this feedback loop system detects that the adaptation goals may be compromised, it needs to determine how to adapt the system to ensure the goals. To that end, the possible options for adaptation are analyzed and the best option that achieves the goals is selected to adapt the system. We refer to the set of possible options for adaptation as the *adaptation space*.

The analysis of adaptation options can be resource and time consuming, in particular if formal analysis is used. Exhaustively analyzing all options may be infeasible for systems with large adaptation spaces [16]. Solving this problem is complex as the adaptation options typically include uncertainty parameters that can only be resolved at runtime, hence the problem is dynamic and needs to be performed during system operation. Different techniques have been proposed to manage the cost of formal analysis at runtime, such as compositional verification [17], pre-computation-based verification [18], and optimization such as caching and look ahead [19]. These techniques reduce the resources and time needed for verification for certain adaptation scenarios and specific types of models and properties.

In this paper, we contribute a complementary approach to tackle the problem of exhaustive analysis of large adaptation spaces using online machine learning. The approach enhances the traditional MAPE-K feedback (Monitor - Analyzer - Planner - Executor - Knowledge) loop with a learning module that supports the analyzer by selecting relevant adaptation options from large adaptation spaces. In particular, the learning module determines at runtime subsets of adaptation options from a large adaptation space, enabling the analyzer to perform efficient analysis. We instantiate the approach for two classic learning techniques: classification and regression, and we evaluate the approaches for two instances of an IoT application for smart environment monitoring with different sizes of adaptation spaces. Given the complexity and high degree of uncertainties, IoT is a particular interesting domain to apply self-adaptation [20]. Empirical evaluation shows that both learning approaches are capable to reduce the adaptation space significantly without noticeable effect on the realization of the adaptation goals.

The remainder of the paper is structured as follows. Section II introduces the IoT application and further refines the problem we tackle in this paper. Section III introduces the machine learning approach for efficient analysis of large adaptation spaces in self-adaptive systems. We instantiate the approach for two learning techniques. In Section IV we evaluate the approach using two instances of the IoT application introduced in Section II. Section V discusses related work on the use of machine learning in self-adaptive systems. Finally, we draw conclusions and look at future research in Section VI.

II. IoT Application and Problem Refinement

In this paper we use two instances of an IoT application for smart environment monitoring to further explain the research problem we tackle and to evaluate the learning approach we propose. Both applications, referred to as DeltaIoT.v1 and

978-1-7281-3368-3/19/$31.00 ©2019 IEEE
DOI 10.1109/SEAMS.2019.00011

DeltaIoT.v2, are developed by KU Leuven and VersaSense[1] and are deployed at the KU Leuven Campus. For practical reasons, we use a simulator for the evaluation of the systems since extensive experimentation on a physical IoT deployment is particularly time consuming. The main differences between the two instances of the application are the deployment and the size of the adaptation spaces, allowing to tackle the problem incrementally. We start this section with a brief introduction of the two instances of the IoT application and we explain the basis architecture of the self-adaptive system. Then we further explain the research problem we tackle in this paper.

A. DeltaIoT.v1

The first IoT application is DeltaIoT.v1 that has been presented as a research artifact for the community [21].

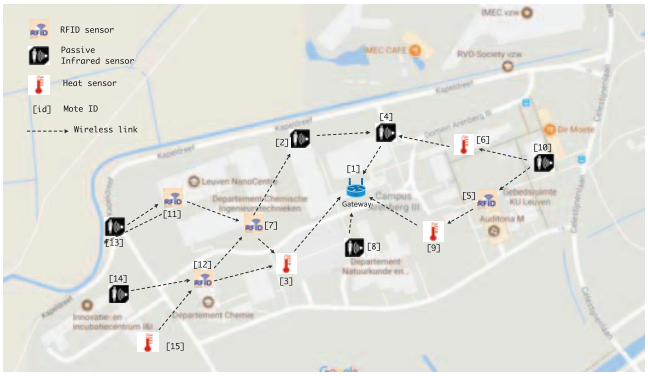

Fig. 1. DeltaIoT.v1 deployment at KU Leuven Campus [21]

DeltaIoT.v1 consists of 15 Long-Range (LoRa) motes that are deployed at the Campus as shown in Figure 1. Each mote is equipped with a sensor that collects data that it relays to the gateway using wireless multi-hop communication. The network includes RFID sensors, passive infrared, and heat sensors. The sensor data is forwarded to an end-user application to monitor the Campus area and take action when needed. Communication is time-synchronized and organized in cycles with slots. Each slot defines a sender and a receiver mote. Motes are battery powered and sending messages dominates the energy cost. For our evaluation, we used a cycle time of 570 seconds (9.5 minutes), each cycle comprising slots that are allocated for communication between two motes. The slots of the first 8 minutes are used for data communication, the remainder slots are available to send messages for adapting the settings of motes.

We consider two types of uncertainties: *network interference* (e.g. due to environment conditions) and fluctuating *load of messages* (e.g., messages generated by motes may depend on the presence of people). As an example, Figure 2 shows the network interference over a period of 100 cycles. We derived these profiles from measurements on the network deployed at the Campus and incorporated the profiles in the simulator.

To deal with these uncertainties, current practice typically applies a conservative approach, where the transmission power of motes is set to maximum and all messages are forwarded to all parents. Operators may tune some settings based on observations and experiences. This approach guarantees high reliability, but implies high energy cost, reducing network lifetime.

[1]www.versasense.com

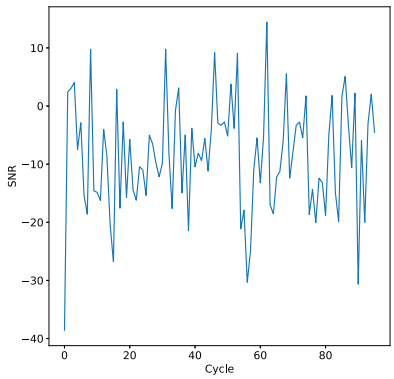

Fig. 2. Example profile of interference over a link

For the evaluation with DeltaIoT.v1, we use three typical quality requirements that need to be realized in IoT, see e.g., [20], [21], regardless of the uncertainties in the network:

- R1: The average packet loss over 12 hours should not exceed 10% of the messages sent;
- R2: The average latency over 12 hours should not exceed 5% of the cycle time;
- R3: The average energy consumption over 12 hours should be minimized.

When self-adaptation is applied, these quality requirements become the adaptation goals. The system provides effectors to adapt the system configuration and realize adaptation; here we use the *power settings* of the motes for communication of messages over links (can be set between 1 for min power and 15 for max power), and the *distribution of the messages* sent by each mote over the links to its parents (can be set in steps of 20%; e.g., for motes with two parents: 0% to one parent and 100% to the other, 20/80, 40/60, 60/40, 80/20 and 100/0). Based on the possible settings of the effectors, the total number of system configurations for DeltaIoT.v1 is 216.

B. DeltaIoT.v2

Recently, we deployed a new instance of the IoT application at the KU Leuven Campus comprising 37 motes distributed across different floors of the main building of the Computer Science Department (see Figure 3), i.e., DeltaIoT.v2.

The underlying technology of DeltaIoT.v2 is similar as the basic DeltaIoT.v1 network, but the adaptation space is much larger and thus more challenging for formal analysis at runtime. For the evaluation of the learning techniques in this paper, we consider the same uncertainties (network interference and load of messages) and adaptation goals as for DeltaIoT.v1 (over 12 hours: average packet loss $\leq 10\%$, average latency $\leq 5\%$ of the cycle time, and average energy consumption minimized). The network of DeltaIoT.v2 is configured with communication cycles of 12 minutes, 9.5 minutes for sending sensor data and 2.5 minutes for sending messages to adapt the motes. We used

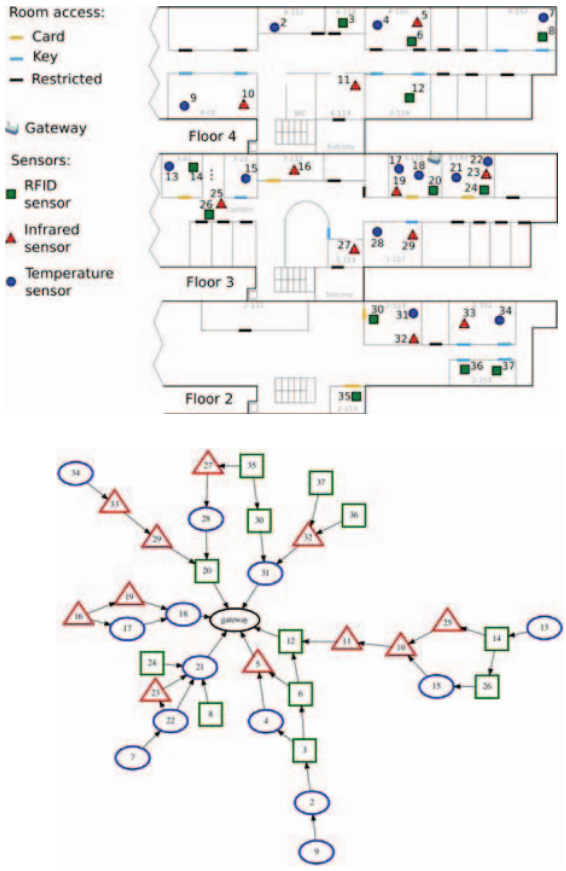

Fig. 3. DeltaIoT.v2: top: physical deployment; bottom: network structure

the same effector for the power setting of the motes (min 1 to max 15), but for the distribution of the messages to multiple parents we used steps of 34% (0/100, 34/66, 66/34, 100/0). Based on the possible settings of the effectors, the total number of adaptation options for DeltaIoT.v2 is 4096.

C. Basic Architecture of the Self-Adaptive IoT System

Figure 4 shows the basic architecture of self-adaptive IoT system (without learning), which is similar for both instances. The managed system is deployed at the level of the gateway. We use a classic MAPE-K feedback loop [12], [22]–[25]. The monitor tracks the IoT system and its environment and updates the corresponding models of the knowledge. The analyzer uses the verifier to analyze the relevant adaptation options, i.e., the verifier typically uses runtime models [26]–[28] of the qualities of interest to determine the expected values of the qualities for each adaptation option. Based on these results, the planner then uses the adaptation goals to select the best option and generate a plan that the executor uses to adapt the IoT system.

For the evaluation of the learning approach, we designed the managing system with ActivFORMS [29] that relies on runtime models of the complete feedback loop that are directly executed by a virtual machine to realize adaptation at runtime. The analyzer in ActivFORMS uses a statistical model checker as verifier [30], [31]. Statistical model checking combines simulation of first-class runtime models with statistical techniques to

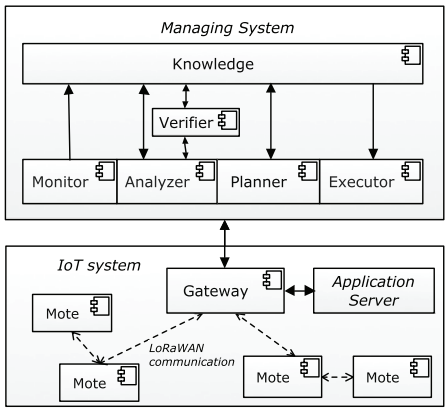

Fig. 4. Basic architecture of self-adaptive IoT system

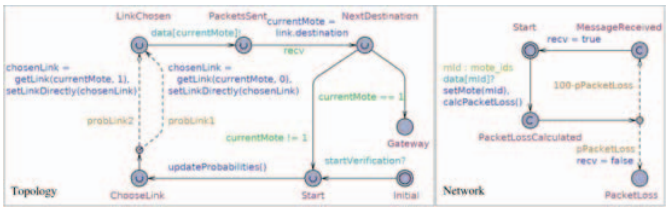

Fig. 5. Runtime model used to predict the packet loss of adaptation options

predict the qualities of each adaptation option with a required accuracy and confidence. As an example, Figure 5 shows the runtime model for packet loss that we use in the evaluation.

After initialization of the adaptation option (not shown), the Topology automaton is triggered to start verification. A link for communication is then selected between two motes based on the probabilities of *probLink1* and *probLink2*. Next, data is sent over the link *data[currentMote]!*, which triggers the Network automaton that calculates the packet loss based on the probability that packets get lost over the link (*pPacketLoss*). This process is repeated for all communication. The verifier then uses a reachability query to determine the probability that packets get lost for the adaptation option (state *PacketLoss*).

For details on ActivFORMS and the use of statistical model checking at runtime, we refer the interested reader to [29], [32].

D. Research Problem

We now further explain the research problem we tackle in this paper. Consider Figure 6 that shows the complete adaptation space for DeltaIoT.v2 at some point in time.

Each dot in this figure represents one adaptation option, i.e., a configuration of the system with a particular setting for the transmission power of each mote for each link, and a setting of the percentage of messages that are sent over each link for motes with multiple parents. For each dot the predicted values for two quality properties of that option are shown (packet loss and latency) that are determined by verification.

As we explained above, for the DeltaIoT.v2 network the adaptation space contains in total 4096 adaptation options. Applying formal verification at runtime to analyze all these options within the available time window (i.e., the cycle time)

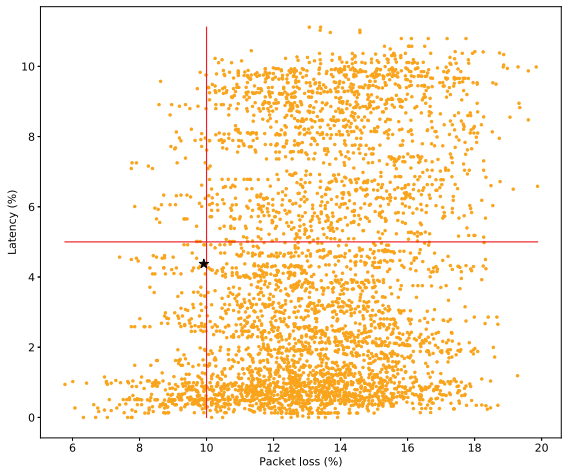

Fig. 6. Adaptation space for DeltaIoT.v2 at some point in time; each dot represents one adaptation option with its predicted values for two qualities.

is not feasible; e.g., when using statistical model checking, verifying the complete adaptation space takes on average 12.5 minutes (for detailed results see the evaluation section). We computed the values for the qualities shown in Figure 6 offline.

In order to efficiently analyze the adaptation space, the analyzer should in principle only consider the "good" adaptation options, i.e., those with expected qualities that comply with the adaptation goals. In Figure 6, these are the adaptation options in the box bounded by the thresholds of the adaptation goals.

It is important to note that the qualities for the adaptation options depend on the actual conditions of the network, i.e., the uncertainties (interference on links, load of traffic). Hence, the adaptation space is dynamic, i.e., the qualities of the adaptation options change over time. Figure 7 shows how the predicted values of two quality properties for a single adaptation option of DeltaIoT.v2 change due to uncertainties over 10 cycles.

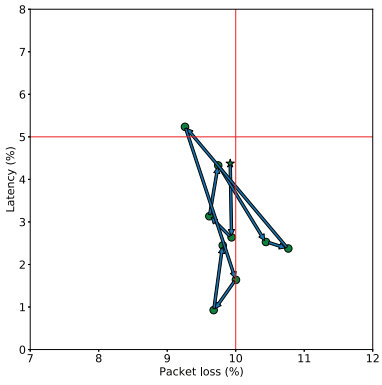

Fig. 7. Movement of an adaptation option due to uncertainties over 10 cycles (the starting position of this option is marked with a black star in Figure 6)

This brings us to the research problem we aim to tackle:

Can machine learning techniques be used to determine subsets of adaptation options from large adaptation spaces allowing a self-adaptive system to perform more efficient analysis without compromising the realization of the adaptation goals?

More efficient means: be able to perform analysis within the available time (in our case the time of one communication cycle). Here we focus on two concrete machine learning techniques when tackling this problem: classification and regression.

III. Machine Learning Approach for Efficient Analysis of Large Adaptation Spaces

We now introduce the online learning approach that we devised to tackle the research problem described in the previous section. We start with a general overview of the approach that shows how learning is integrated in the feedback loop system of a self-adaptive system. Then we instantiate the general approach for classification and regression respectively.

A. Overview of the Learning Approach

To support the analysis stage of adaptation, we equip the feedback loop system with a machine learning module. The task of the learning module is to determine a set of relevant adaptation options for the analyzer by exploiting the available analysis results. Figure 8 shows how the machine learning module is integrated within the feedback loop work flow.

The *Monitor* tracks relevant uncertainties and properties of the managed system and its environment (1) and updates the corresponding models in the *Knowledge* repository (2). When the *Analyzer* is triggered (3) it collects the adaptation options from the repository (4). The adaptation options define the configurations of the system that can be reached by adapting the current configuration. The task of the analyzer is to predict the qualities that are subject of adaptation for a relevant set of adaptation options given the actual operating conditions (uncertainties). To that end, the analyzer triggers the *Machine learner* to determine a relevant set of adaptation options (5.1). The machine learner comprises a learning algorithm that uses a learning model. A learning model (also referred to as hypothesis) comprises the target function that we want to learn. The target function allows mapping input to output, e.g., mapping adaptation options to classes that define the compliance of the adaptation options to the adaptation goals. The learning algorithm is a set of instructions that tries to model the target function using input data based on a set of possible hypotheses. The machine learner identifies a subset of adaptation options that, based in its current knowledge, are expected to comply with the adaptation goals (5.2). The analyzer then invokes the *Verifier* to determine the qualities of the set of adaptation options determined by the learner (6.1). For this, the verifier analyzes runtime models of the quality properties for the selected adaptation options; the runtime models are maintained in the knowledge repository (6.2). Once the verification step is done, the analyzed adaptation options are updated, i.e., the analyzer associates the verification results with each of the selected adaptation options (7). The analyzer then triggers the machine learner to exploit the novel analysis results (8.1), continuing the learning process. As a result, the learner updates the learning models using the analysis results (8.2). When the *Planner* is triggered (9) it determines the best adaptation option (10) based on the adaptation goals, using a decision making mechanism such as a set of rules or a utility function. The planner then creates a plan to adapt the

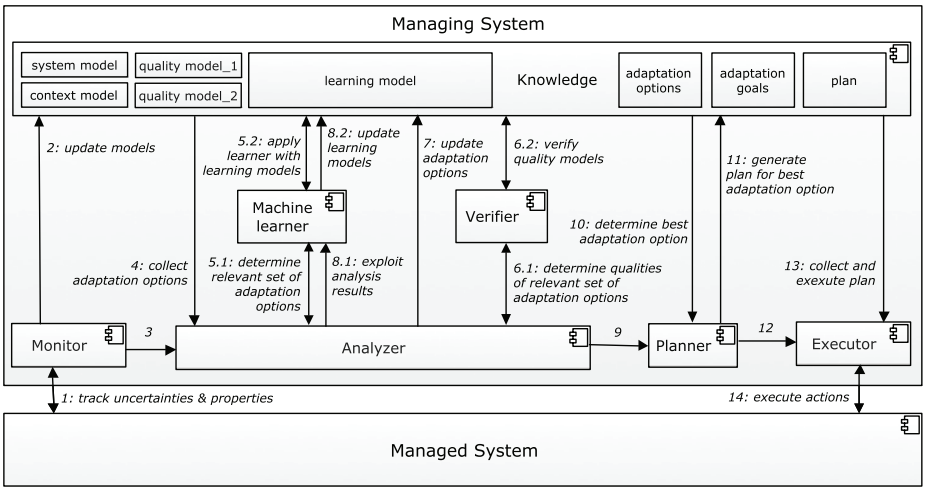

Fig. 8. Architecture of the generic machine learning approach to reduce large adaptation spaces

managed system according to the best adaptation option (11). Finally, the *Executor* (12) collects the plan and executes it (13) by enacting the adaptation actions to the managed system (14).

B. Instantiation of the Approach using Classification

The learning approach can be instantiated using any suitable online learning mechanism. Here we consider two instantiations: classification and regression, two mature supervised learning approaches that are well studied and easy to use. In this paper, we apply the learning approaches to reduce the adaptation space of a self-adaptive system based on two threshold goals (concretely: average packet loss $\leq 10\%$, average latency $\leq 5\%$ of the cycle time).

We start with classification [33], which is a classic supervised learning technique that aims at assigning an observation (in our case an adaptation option) to a labeled class from a predetermined set of classes (here classes are defined by compliance with zero, one, or two adaptation goals). For self-adaptive systems, classification can be used to predict whether an adaptation option fulfills the adaptation goals. The adaptation options that comply with all the adaptation goals are labeled relevant by the classifier and will be analyzed by the verifier. From the adaptation options that are classified as compliant with only one of the adaptation goals, a small fraction will be analyzed. This analysis allows exploring new adaptation options that were classified as irrelevant before, but may have become relevant due to changes in the uncertainties (cf. Figure 7).

Standard classification utilizes batch learning to train its model. The trained model enables mapping observations to classes. Batch learning implies that the classifier is trained once with the complete collection of training data. However, this approach is not sufficient for self-adaptive systems. Uncertainties encountered at runtime are unknown beforehand. The solution to this problem is *online learning*, in particular *incremental learning* that allows a learner to learn more than once, updating the learning model without discarding old training data. In the context of our problem, this is crucial to make sure that predictions are adjusted to newly discovered uncertainties.

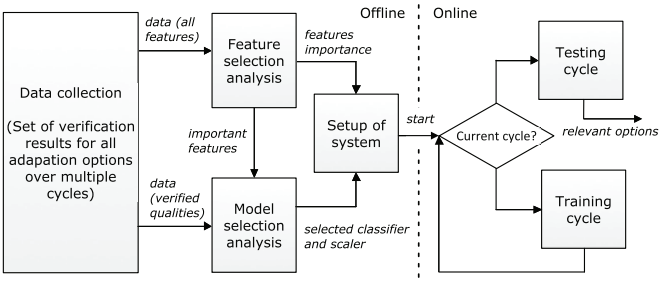

Fig. 9. Work flow of classification with offline and online activities

For both DeltaIoT.v1 and DeltaIoT.v2, we apply a linear classifier that uses Stochastic Gradient Descent (SGD); in short a SGD classifier.[2] This classifier estimates the gradient of the loss for each observation it receives by comparing the outcome predicted by the learner with the input sample (i.e., the learner predicts the class an adaptation option belongs to and compares this with the results obtained by the verifier). The loss represents the inaccuracy of predicting the category a particular observation belongs to. The model is then updated based on the estimated loss, improving the model and thus the target function. The SGD classifier showed to be most suited for both instances of the IoT system when we compared different classifiers using different quality metrics. We further discuss this in the valuation in Section IV.

Figure 9 shows the work flow of classification that consists of a number of offline activities (done before the system is running) and a number of online activities (done at runtime).

1) Offline Activities: The first offline activity is collecting data that is required for a number of analytical tasks that need to be performed to set up the learning module. In our context, we need data of verification results for the adaptation options over a number of cycles. These results will contain data of the *verified qualities* and data of the *features*. Features are variables

[2]https://scikit-learn.org/stable/modules/sgd.html

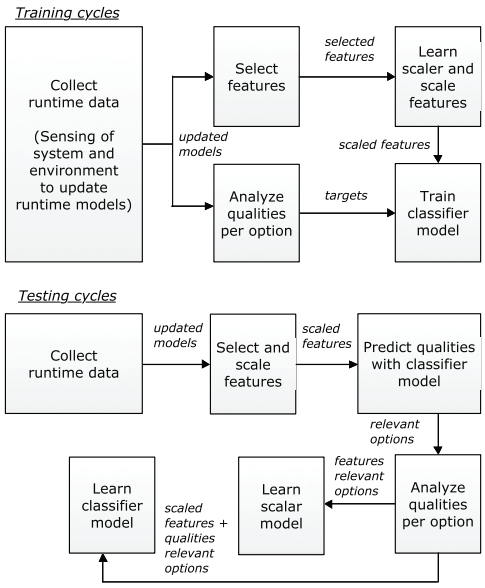

Fig. 10. Online work flow for classification with training and testing cycles

in the system that affect the predicted qualities of adaptation options, incl. the uncertainties (network interference and load) and effector settings (power settings of motes to communicate over links and distribution of messages sent to parents).

Using this data, the second activity that is required is feature selection using a feature selection algorithm. Feature selection evaluates the effects that different features have on the actual prediction of the classifier. The effect of the features is expressed by giving each feature an importance score. Features with a very low score are considered not relevant to the predictions of the classifier. In this work, we have filtered out these features, so they are not used at runtime by the classifier. Feature filtering helps simplifying the model of the classifier and empirical evaluation showed that it also improved the accuracy of the classifier. A potential disadvantage is that we may have missed relevant features due to the specific coverage of the training set.

The third offline activity is model selection, where multiple combinations of classifiers (with variations in their loss and penalty functions)[3] and scalers (which transform features to a specific scale) are evaluated. It is important to note that this evaluation is done with only the relevant features determined by feature selection as mentioned before. During model selection the designer evaluates different combinations of classifiers and scalers using a number of quality metrics (including accuracy and F1-score) that we further discuss in the evaluation section. The result of this activity is a selected classifier and scaler that is used to support efficient runtime analysis.

Once the relevant features and the selected classifier and scaler are known, the machine learner module and corresponding models can be instantiated and integrated in the feedback loop system (as in Figure 8). This concludes the activities that are required before learning can be used at runtime.

[3]Whereas a loss function corresponds to the inaccuracy of predictions as explained above; a penalty function determines the degree of impact that the loss will have on the model of the learner.

Applied to DeltaIoT.v1, for the evaluation of this network, we make use of a standard scaler and a SGD classifier. The standard scaler scales the features according to their respective means and standard deviations observed so far, which provides the bests results for the simple network. The scaler keeps track of the means and standard deviations for each feature and stores this information in the scaler model. At runtime, the scaler updates the means and standard deviations values taking into account new data. As explained above, the SGD classifier is a linear classifier that uses stochastic gradient descent as optimization method to learn and improve its model.

For the evaluation of DeltaIoT.v2, we use a minimum-maximum scaler and SGD classifier. A minimum-maximum scaler scales features, as its name suggests, according to their minimum and maximum values encountered over time. The scaler maps the values of the features to the interval [0,1]. We use the same classifier for this network as for DeltaIoT.v1, but with adjusted functions for loss and penalty (based on evaluation during model selection, as explained above).

2) Online Activities: When the self-adaptive system is running, we differentiate between two activities corresponding to the training and learning cycles shown in Figure 10. These activities are performed completely automatically.

During the training cycles, runtime data of the system and the environment is collected and used to update the runtime models. On the one hand, this novel runtime knowledge is used to select and scale relevant features. On the other hand, the up to date models are used to formally analyze adaptation options. The relevant features of the analyzed adaptation options and their respective qualities are then used to train the classifier model. During training, the adaptation space is not reduced yet since the classifier is not sufficiently trained. Training of the classifier model can be based on the analysis results of a selection of adaptation options per cycle, or training can run as a background process before testing starts.

In the testing cycles, the predictions of the classifier are used to select subsets of adaptation options for further analysis by the feedback loop. The selected sets of relevant adaptation options are those that are predicted by the classifier to meet both adaptation goals. As explained above, a small sample of additional adaptation options that are classified as compliant with only one adaptation goal are analyzed as well in order to explore new adaptation options under changing operating conditions. The features of the relevant adaptation options together with the quality predictions for the set of analyzed adaptation options are used to continue the learning of the classifier and its models. This way, the classifier is able to adjust earlier predictions using the novel knowledge discovered and generated at runtime. Such adjustments may be required for potentially mis-predicted classifications and to exploit correlations between uncertainties and qualities that were not known before.

Applied to the IoT network instances, we train the classifier model during a number of cycles that are empirically determined during the model selection process. For DeltaIoT.v1, we train the classifier model using verification results of the analysis of the complete adaptation space as this is feasible within the time of a cycle for this configuration. For DeltaIoT.v2, we use the

verification results of a subset of all adaptation options in each cycle that can be verified within the available time slot to train the classifier model. The selection of subsets is done such that the complete adaptation space is incrementally analyzed over a number of cycles and this repeated over the training period.

3) Architecture Instantiation for Classification: Figure 11 shows the main part of the generic runtime architecture with learning instantiated for classification. The figure summarizes the different steps of the testing phase explained above.

C. Instantiation of the Approach using Regression

Regression is a supervised learning technique that aims at predicting the values of dependent variables based on observations [34]. Similar to classification, the task of regression is to approximate a target function (or mapping function) from input data to an output variable, which is for regression a continuous variable (compared to a discrete class label for classification). In our context, the output variable represents expected values of qualities of adaptation options of the system. As for classification, the values predicted by a regressor can be used to reduce the adaptation space to a subset of adaptation options that are deemed relevant for analysis. Since regression predicts values, the approach is more easily to use when dealing with optimization goals compared to classification, but in this paper we only apply learning for threshold goals. Similar to classification, standard regression uses batch learning to train its model. Since we are interested in *online learning*, we only considered regressors that support this type of learning.

1) Offline and Online Activities: The offline and online activities for regression are almost identical as for classification; see Figure 9 in Section III-B1, and Figure 10 in Section III-B2. The main differences lie in the selection of a regressor and its model (offline) and the way relevant adaptation options are selected in the testing phase (online). Instead of looking at the predicted classes from classification, we now look at the predicted values and compare these values to the user goals.

As the aim of learning in our case is to select subsets of adaptation options based on two adaptation goals that are defined by thresholds (average packet loss and average latency), we converted the predicted values retrieved from regression to the corresponding classification classes. E.g., a predicted latency value of 3% and a predicted packet loss value of 8% for a particular adaptation option is converted to a predicted class that meets both goals. Consequently, we have also used the same metrics in the model selection process to choose the most suited regressors and scalers for both networks.

Applied to DeltaIoT.v1, we use a combination of a minimum-maximum scaler and a SGD regressor. For DeltaIoT.v2 we use a combination of a standard scaler and a SGD regressor. These combinations provided the best learning results. We explained the scalers above. The SGD regressor is similar to the SGD classifier and applies linear regression using stochastic gradient descent optimization to minimize the loss of the model.

2) Architecture Instantiation for Regression: The concrete architecture of the learning module for regression is similar to the one for classification as shown in Figure 11. The stochastic gradient decent classifier and the standard scaler of the machine learner of Figure 11 now become the stochastic gradient decent regressor and the minimum-maximum scaler respectively. Similarly, the classifier model and the standard scaler model that are part of the knowledge in Figure 11 now become the regressor model and minimum-maximum scaler model.

IV. Evaluation

We evaluate the learning approaches with the two IoT applications described in Section II. For each application, we discuss the concrete setup, the offline model selection process, and the online results obtained by applying adaptation with the learning approaches. For the adaptation results, we compare the results with a reference approach that analyses the whole adaptation space (i.e., without learning). For the learning approaches, we used the implementations of the classifiers, regressors, and scalers from *scikit-learn* [35]. The tests are performed using a simulator of IoT applications. This simulator together with all the test results are available online.[4]

A. Evaluation with DeltaIoT.v1

1) Setup: For the evaluation with DeltaIoT.v1, we used a setup with 15 motes as shown in Figure 1. We used uncertainty profiles for loads of messages (0...10 msgs per mote per cycle) and network interferences (-40dB...+15dB), similar to the profile shown in Figure 2. Recall that the total number of adaptation options for DeltaIoT.v1 is 216. We designed the MAPE feedback loop system using a network of timed automata models. These models are directly executed to realize adaptation using the ActivFORMS execution engine [36]. For the analysis of adaptation options we applied runtime statistical model checking using Uppaal-SMC [30].

To evaluate the online learning approaches, we simulate the running self-adaptive system for a period of five days (from 6AM to 6PM). We run the simulator on a computer system with an *i7 8550U 1.8 GHz* processor and *8GB RAM*.

The learners learn models for two adaptation goals defined by thresholds (packet loss and latency) as explained in Section II. We use 4 possible classes to classify the adaptation options:

- C_0: No goals are satisfied.
- C_1: Only the packet loss goal is satisfied.
- C_2: Only the latency goal is satisfied.
- C_3: Both goals are satisfied.

For regression, we use the predicted values for both qualities to evaluate whether adaptation options meet the packet loss and latency goals. These are then converted to one of the classes.

Algorithm 1 Selection of adaptation options for verification

```
1: predictions = classifier.makePredictions(allOptions)
2: explored = predictions.getRandomSet(predictions.class(1,2), 0.05)
3: subset = predictions.class(3) + explored
4: analyzer.verifyAdaptationOptions(subset)
```

The learning process results in a selection of a subset of relevant adaptation options in each cycle that is considered for analysis, as described in Algorithm 1. Firstly, predications are made for all adaptation options (line 1). A subset of adaptation

[4]https://people.cs.kuleuven.be/danny.weyns/software/machine-learning.htm

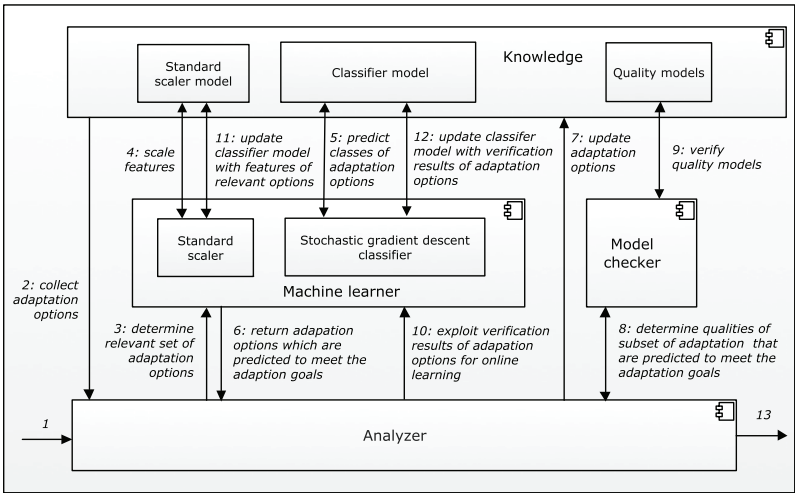

Fig. 11. Concrete instantiation of the main part of the general architecture for classification

options, which are predicted to only comply with one goal, is selected for exploration (line 2). Lastly the relevant options (predicted to be C_3) and an exploration subset are verified. For each of these adaptation options, the verifier predicts the values of the qualities for the three adaptation goals. The option with packet loss and latency below the thresholds and with minimum energy consumption is then selected for adaptation.

2) Model selection process: For the offline activities, we collected data over a period of one week (6AM to 6PM). This corresponds to data of 300 cycles with verified qualities and verification times of the adaptation options. The required amount of data was determined empirically based on the quality of the results obtained using learning. With the data we determined the classifier and scaler for both instances of the IoT application.

During model selection, we considered a number of different types of classifiers/regressors and scalers. Specifically, we have covered most classifiers and regressors that support online learning from scikit learn, as well as the scalers that support online learning. With the data we collected, we simulated different classifiers and regressors and applied a number of metrics to the prediction results. In particular, we considered the following commonly used metrics: overall accuracy, F1 score and Matthews correlation coefficient.

The overall accuracy describes the accuracy of the results predicted in the testing cycles. We measured accuracies from 20% to 80% for the evaluated classifier/regressors and scalers.

Next, we evaluated the models based on their F1 score. This metric combines recall (the percentage of samples that were retrieved by prediction from a specific class) and precision (the fraction of samples that have been predicted to be of a specific class that are actually part of that class). The F1 score is defined as a value in [0,1]. A higher F1 score means in general a better performing classifier. For F1, we looked at a combined score which covers all the individual classes, and at separate F1 scores per class (C_0...C_3). The F1 score (C_3) ranged from 0.2 to 0.9 for the different combinations of classifier/regressors and scalers.

Lastly, we looked at the Matthews correlation coefficient. This metric produces a value in the range [-1,1], where 1 refers to perfect predictions, 0 means predictions that are equal to random, and -1 refers to completely incorrect predictions. The Matthews correlation coefficient ranged from -0.01 to 0.7 for the different combinations of classifier/regressors and scalers.

Table I summarizes the results for the selected classifiers and regressors (Classif v1 and Regres v1 apply to DeltaIoT.v1).

	Classif v1	Regres v1	Classif v2	Regres v2
Accuracy	79.93%	81,43%	86.81%	84.85%
Matthew's cor.	0.6565	0.6776	0.7003	0.6500
F1-score (comb)	0.7922	0.7823	0.8624	0.8381
F1-score (C_3)	0.8585	0.8843	0.7301	0.7350

TABLE I
METRICS FOR THE CLASSIFIERS (CLASSIF) AND REGRESSORS (REGRES) OF DELTAIOT.V1 (V1) AND DELTAIOT.V2 (V2)

3) Online Adaptation Results: As explained in Section III, the learners first go through a number of training cycles. For DeltaIoT.v1 we applied 45 training cycles. This number was empirically determined during the model selection process.

Figure 12 shows the results for online adaptation with both learning techniques and the reference approach. The results show that there is no significant difference in the results between all approaches for the two quality goals that are used to reduce the adaptation space, packet loss and latency. The same applies to the results for energy consumption. The p-values for the different qualities based on the Wilcoxon signed-rank test for all the results are between 0.51 and 1.00. On the other hand, the time required for verifying the adaptation options is significantly lower with the learning approaches compared to the reference approach (mean values: 31.80 s for no learning, 14.51 s for classification, and 13.98 s for regression; we did not find a significant difference between classification and regression with p-value 0.98). The time for the learning approaches shown in the graph is a combination of the learning time (predicting and selecting adaptation options + online learning) and the verification time. We observed that the learning time for both approaches is a small fraction of the total time spent ($< 1\%$). Note that the outliers for time with the learning approaches represent the time required for training.

B. Evaluation with DeltaIoT.v2

1) Setup: For the evaluation with DeltaIoT.v2, we used a setup with 37 motes as shown in Figure 3. We used similar uncertainty profiles as for DeltaIoT.v1 and apply learning for the same adaptation goals. The total number of adaptation options for DeltaIoT.v2 is 4096, also explained in Section II. We used a similar MAPE feedback loop system as for DeltaIoT.v1, and for the analysis of adaptation options we applied runtime statistical model checking as well.

To evaluate the online learning approaches, we simulate the running self-adaptive system for a period of four days (from 6AM to 6PM). We run the simulator on a stronger computer system with an *intel xeon skylake 8 core 2GHz* machine with *16GB RAM*.

We use the same four classes to classify the adaptation options as for DeltaIoT.v1 (C_0 to C_3). We transfer the predicted values with regression again to these classes and we use Algorithm 1 to select adaptation options for verification.

2) Model selection process: For the offline activities of DeltaIoT.v2, we collected verification data over a period of one week corresponding to 300 cycles, and we followed the same procedure for model selection as for DeltaIoT.v1. Table I summarizes the results of the metrics for the selected classifiers and regressors (Classif v2 and Regres v2 apply to DeltaIoT.v2).

3) Online Adaptation Results: For DeltaIoT.v2 we trained the learning models during 60 cycles, which was empirically determined during the model selection.

Figure 13 shows the adaptation options for one particular cycle that the regressor predicted to be of class C_3. The figure also shows the adaptation options of other classes that were selected for verification in this cycle.

Figure 14 shows results for online adaptation with both learning techniques and the reference approach. It is important to note that the results for the reference approach were obtained via offline verification as exhaustive verification is not feasible within the available time slot of a cycle. The results confirm that there is no significant difference in the results between all approaches for the three quality goals. The p-values for the different qualities based on the Wilcoxon signed-rank test for all the results are between 0.03 and 1.00. The time required for verifying the adaptation options when learning is applied is also for DeltaIoT.v2 significantly lower compared to the reference approach (mean 747.44 s for no learning, 174.50 s for classification, and 167.97 s for regression; in this case the time for regression was significantly lower as for classification with $p = 0.0004$). The learning time remains very small compared to the verification time ($< 1\%$). The outliers for time with learning (approximately 570 seconds) represent here also the time required for training.

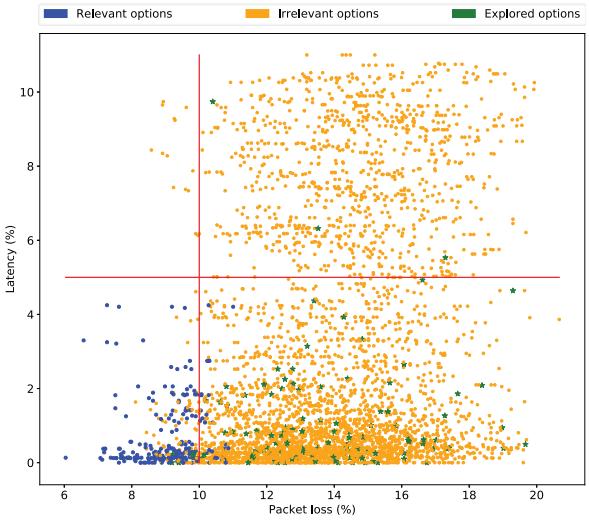

Fig. 13. Selected/explored adaptation options with regression in one cycle

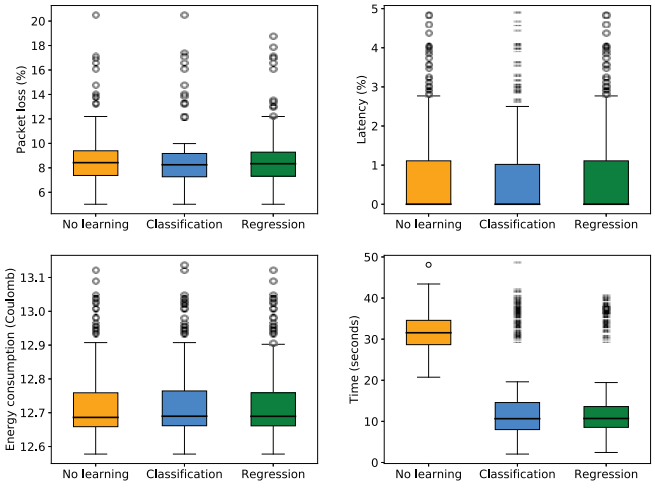

Fig. 12. Adaptation results for the DeltaIoT.v1 with and without learning

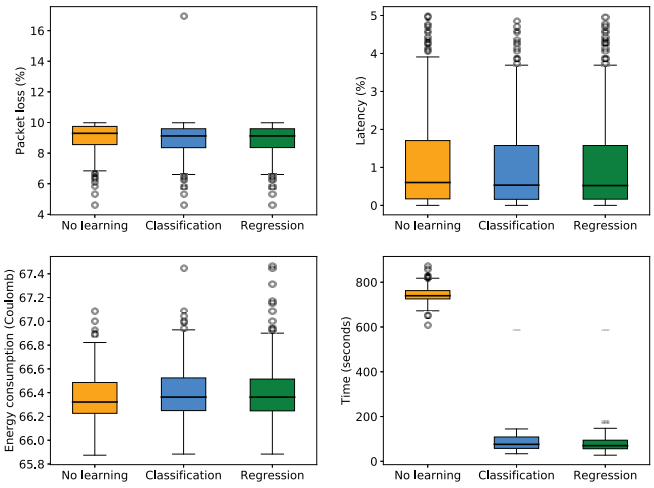

Fig. 14. Adaptation results for the DeltaIoT.v2 with and without learning

C. Conclusion of Evaluation

The research question we aimed to answer in this paper was: Can machine learning techniques be used to determine subsets of adaptation options from large adaptation spaces allowing a

self-adaptive system to perform more efficient analysis without compromising the realization of the adaptation goals?

The evaluation results demonstrate that we can answer this question positively. The learning approaches reduce the adaptation space on average with 50.0% for DeltaIoT.v1 and 75.5% for DeltaIoT.v2. This reduction results in an average time reduction of 55.7% and 76.6% respectively, without compromising the realization of any of the adaptation goals (we could not measure a significant difference for any of the quality properties).

D. Threats to Validity

The evaluation of the learning approaches is subject to a number of validity threats. Since we applied the learning approach to the particular domain of IoT, we cannot generalize the conclusion (external validity). We tried to anticipate this threat to some extent by applying two different learning techniques to two different IoT networks (in terms of topology and size of adaptation space). But obviously, more extensive evaluation in different domains is required to obtain further insight.

We have evaluated the ability of the learning approach to reduce the adaptation space by measuring the impact on the quality properties of two concrete applications. The specifics of these applications, in particular the network structure, the types of uncertainties, and the specific goals that we considered may have an effect on the difficulty to demonstrate the benefits of the proposed learning approach (internal validity). We mitigated this threat to some extent by considering real application settings that are proposed by an industry partner.

The approaches are evaluated in simulation for pragmatic reasons. This simulation includes certain forms of randomness in the parameters of the setup of the applications that correspond to uncertainties. This may introduce a threat that the results may not be the same if the study would be conducted again (reliability). In order to minimize this threat, we applied to learning approaches over a period of several days that represent a large number of adaptation cycles. In addition, the complete evaluation package is available to replicate the study.

V. Related Work

In the introduction, we highlighted a number of approaches that have been proposed to manage the cost of formal analysis at runtime, including [17], [18], and [19]. These approaches apply techniques to improve the verification process. We apply learning as a complementary approach that aims at reducing the adaptation space. In this section we look at a number of related approaches that apply learning in self-adaptation with a focus on supervised learning as we applied in our approach.

Epifani et al. [37] presents a pioneering work on learning for self-adaptive systems. The authors use probabilistic quality models that include uncertainties as model parameters. The parameters are initially set based on expert knowledge. Then, runtime data is collected and fed to a Bayesian estimator that updates the model parameters. The approach was used in [38] to support quantitative verification at runtime. Our work has a different focus, but the approach of [38] can be used in our work to keep quality models up to date.

Rodrigues et al. [39] exploit knowledge obtained from offline verification and simulations of a prototype to provide assurances for real-time adaptive systems at runtime. In addition, data mining and classification are used at runtime to learn prediction rules that are used to manage resources and decision trees to support the decision making process to conform to the verified properties in different contexts. In contrast, our approach learns the relevant adaptation options under uncertainty that are then analyzed using model checking to make adaptation decisions.

Duarte et al. [40] propose a method to learn human-readable models that explicitly capture uncertainties. These models can be used by human operators in-the-loop. The approach relies on K-plane clustering to learn possible outcomes of an action on the system quality and probabilities associated to each outcome. Expert-defined models are used to accelerate the learning process. We apply learning to enhance the efficiency of automatic formal analysis in the feedback loop work flow.

FUSION [41] applies learning to determine the impact of adaptation decisions on the system's goals. To that end, the authors apply the M5 model tree algorithm to discover relationships between features (i.e., system capabilities) and metrics (i.e., measurable quantities, e.g., response time). As our approach, FUSION is able to improve analysis significantly, but whereas FUSION targets the feature selection space, our work targets the configuration space.

Ghahremani et al. [42] uses Random Forest, Gradient Boosting Regression, and Extreme Boosting Trees, to predict changes of the system utility without knowing the details of the system. The approach predicts the quality attribute values and then sorts the adaptation options based on the differences between previous and current prediction values. Our approach uses learning to select a subset of the adaptation options which are then verified by the model checker to select the optimal option.

VI. Conclusions and Outlook

In this paper, we investigated whether machine learning techniques can be used to reduce large adaptation spaces of self-adaptative systems without negatively affecting the realization of the adaptation goals. Our research confirmed that this is indeed the case. We contribute an online learning approach that seamlessly integrates with the standard MAPE-K loop. This approach enables a machine learning component to select a subset of adaptation options for self-adaptation enabling efficient and effective analysis. We demonstrated this for two learning techniques applied to two instances of an IoT application.

Our future work is planned in three directions. First, we plan to explore learning to deal with more than two goals, include optimization goals (e.g., minimizing energy usage in the IoT cases), and changing goals. Second, we plan to study other learning techniques to tackle the problem of adaptation space reduction, e.g. deep learning. Third, we plan to investigate the theoretic bounds on the guarantees for goals that can be achieved when online learning is applied to reduce the adaptation space of a self-adaptive system. [43] provides some inspiring ideas here.

REFERENCES

[1] N. Esfahani, E. Kouroshfar, and S. Malek, "Taming uncertainty in self-adaptive software," in *19th Symposium and the 13th European Conference on Foundations of Software Engineering*, pp. 234–244, ACM, 2011.

[2] D. Perez-Palacin and R. Mirandola, "Uncertainties in the modeling of self-adaptive systems: A taxonomy and an example of availability evaluation," in *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, (New York, NY, USA), pp. 3–14, ACM, 2014.

[3] S. Mahdavi-Hezavehi, P. Avgeriou, and D. Weyns, "A classification framework of uncertainty in architecture-based self-adaptive systems with multiple quality requirements," in *Managing Trade-Offs in Adaptable Software Architectures*, pp. 45–77, 01 2017.

[4] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," *Commun. ACM*, vol. 55, no. 9, pp. 69–77, 2012.

[5] G. Tamura, N. M. Villegas, H. A. Müller, J. P. Sousa, B. Becker, G. Karsai, S. Mankovskii, M. Pezzè, W. Schäfer, L. Tahvildari, and K. Wong, *Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems*, pp. 108–132. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[6] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl, "Proactive self-adaptation under uncertainty: A probabilistic model checking approach," in *Foundations of Software Engineering*, pp. 1–12, ACM, 2015.

[7] A. Filieri, G. Tamburrelli, and C. Ghezzi, "Supporting self-adaptation via quantitative verification and sensitivity analysis at run time," *IEEE Transactions on Software Engineering*, vol. 42, pp. 75–99, Jan. 2016.

[8] D. F. Mendonça, G. N. Rodrigues, R. Ali, V. Alves, and L. Baresi, "GODA: A goal-oriented requirements engineering framework for runtime dependability analysis," *Information & Software Technology*, vol. 80, pp. 245–264, 2016.

[9] J. Cámara, R. de Lemos, and N. L. et al., "Robustness-driven resilience evaluation of self-adaptive software systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 1, pp. 50–64, 2017.

[10] P. Arcaini, E. Riccobene, and P. Scandurra, "Formal design and verification of self-adaptive systems with decentralized control," *TAAS*, vol. 11, no. 4, pp. 25:1–25:35, 2017.

[11] R. Calinescu, D. Weyns, S. Gerasimou, M. U. Iftikhar, I. Habli, and T. Kelly, "Engineering trustworthy self-adaptive software with dynamic assurance cases," *IEEE Trans. Software Eng.*, vol. 44, no. 11, pp. 1039–1069, 2018.

[12] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, pp. 41–50, Jan. 2003.

[13] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. D. M. Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, "Software engineering for self-adaptive systems: A research roadmap," in *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, pp. 1–26, 2009.

[14] R. de Lemos, D. Garlan, C. Ghezzi, H. Giese, J. Andersson, M. Litoiu, B. R. Schmerl, D. Weyns, L. Baresi, N. Bencomo, Y. Brun, J. Cámara, R. Calinescu, M. B. Cohen, A. Gorla, V. Grassi, L. Grunske, P. Inverardi, J. Jézéquel, S. Malek, R. Mirandola, M. Mori, H. A. Müller, R. Rouvoy, C. M. F. Rubira, É. Rutten, M. Shaw, G. Tamburrelli, G. Tamura, N. M. Villegas, T. Vogel, and F. Zambonelli, "Software engineering for self-adaptive systems: Research challenges in the provision of assurances," in *Software Engineering for Self-Adaptive Systems III. Assurances - International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers*, pp. 3–30, 2013.

[15] D. Weyns, "Software engineering for self-adaptive systems," in *Handbook of Software Engineering*, Springer, 2019.

[16] D. Weyns, N. Bencomo, and R. Calinescu et al., "Perpetual assurances for self-adaptive systems," in *Software Engineering for Self-Adaptive Systems III. Assurances*, pp. 31–63, Springer International Publishing, 2017.

[17] R. Calinescu, S. Kikuchi, and K. Johnson, "Compositional reverification of probabilistic safety properties for large-scale complex it systems," in *Large-Scale Complex IT Systems. Development, Operation and Management*, Springer Berlin Heidelberg, 2012.

[18] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, (New York, NY, USA), pp. 341–350, ACM, 2011.

[19] S. Gerasimou, R. Calinescu, and A. Banks, "Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, (New York, NY, USA), pp. 115–124, ACM, 2014.

[20] D. Weyns, G. S. Ramachandran, and R. K. Singh, "Self-managing internet of things," in *SOFSEM 2018: Theory and Practice of Computer Science* (A. M. Tjoa, L. Bellatreche, S. Biffl, J. van Leeuwen, and J. Wiedermann, eds.), (Cham), pp. 67–84, Springer International Publishing, 2018.

[21] M. U. Iftikhar, G. S. Ramachandran, P. Bollanse, D. Weyns, and D. Hughes, "Deltaiot: A self-adaptive internet of things exemplar," in *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2017.

[22] S. Dobson, S. Denazis, and A. Fernández et al., "A survey of autonomic communications," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 1, no. 2, pp. 223–259, 2006.

[23] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 2, pp. 14:1–14:42, 2009.

[24] D. Weyns, U. Iftikhar, and J. Soderland, "Do External Feedback Loops Improve the Design of Self-Adaptive Systems? A Controlled Experiment," in *Software Engineering for Adaptive and Self-Managing Systems*, 2013.

[25] T. Vogel and H. Giese, "Model-driven engineering of self-adaptive software with eurema," *ACM Trans. Auton. Adapt. Syst.*, vol. 8, no. 4, 2014.

[26] G. Blair, N. Bencomo, and R. B. France, "Models@ run.time," *Computer*, vol. 42, pp. 22–27, Oct. 2009.

[27] A. Bennaceur, R. France, G. Tamburrelli, T. Vogel, P. J. Mosterman, W. Cazzola, F. M. Costa, A. Pierantonio, M. Tichy, M. Akşit, P. Emmanuelson, H. Gang, N. Georgantas, and D. Redlich, *Mechanisms for Leveraging Models at Runtime in Self-adaptive Software*, pp. 19–46. Cham: Springer International Publishing, 2014.

[28] B. H. C. Cheng, K. I. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. A. Müller, P. Pelliccione, A. Perini, N. A. Qureshi, B. Rumpe, D. Schneider, F. Trollmann, and N. M. Villegas, *Using Models at Runtime to Address Assurance for Self-Adaptive Systems*, pp. 101–136. Cham: Springer International Publishing, 2014.

[29] M. U. Iftikhar and D. Weyns, "Activforms: Active formal models for self-adaptation," in *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ACM, 2014.

[30] A. David, K. Larsen, and A. Legay et al., "Uppaal smc tutorial," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 4, pp. 397–415, 2015.

[31] A. Legay, B. Delahaye, and S. Bensalem, "Statistical model checking: An overview," in *Runtime Verification* (H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, and N. Tillmann, eds.), (Berlin, Heidelberg), pp. 122–135, Springer, 2010.

[32] D. Weyns, M. U. Iftikhar, D. Hughes, and N. Matthys, "Applying architecture-based adaptation to automate the management of internet-of-things," in *Software Architecture - 12th European Conference on Software Architecture, ECSA 2018, Madrid, Spain, September 24-28, 2018, Proceedings*, pp. 49–67, 2018.

[33] S. B. Kotsiantis, "Supervised machine learning: A review of classification techniques," in *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*, (Amsterdam, The Netherlands), pp. 3–24, IOS Press, 2007.

[34] P. A. Flach, "Machine learning: The art and science of algorithms that make sense of data," *Cambridge University Pres*, 2012.

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[36] M. U. Iftikhar, J. Lundberg, and D. Weyns, "A model interpreter for timed automata," in *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, 2016.

[37] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Model evolution by run-time parameter adaptation," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, (Washington, DC, USA), pp. 111–121, IEEE Computer Society, 2009.

[38] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli, "Dynamic qos management and optimization in service-based systems," *IEEE Transactions on Software Engineering*, vol. 37, pp. 387–409, May 2011.

[39] A. Rodrigues, R. D. Caldas, G. N. Rodrigues, T. Vogel, and P. Pelliccione, "A learning approach to enhance assurances for real-time self-adaptive systems," *arXiv preprint arXiv:1804.00994*, 2018.

[40] F. Duarte, R. Gil, P. Romano, A. Lopes, and L. Rodrigues, "Learning non-deterministic impact models for adaptation," in *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*, pp. 196–205, ACM, 2018.

[41] A. Elkhodary, N. Esfahani, and S. Malek, "Fusion: a framework for engineering self-tuning self-adaptive software systems," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 7–16, ACM, 2010.

[42] S. Ghahremani, C. M. Adriano, and H. Giese, "Training prediction models for rule-based self-adaptive systems," in *2018 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 187–192, IEEE, 2018.

[43] P. Domingos, "A few useful things to know about machine learning," *Commun. ACM*, vol. 55, pp. 78–87, Oct. 2012.