Learning Utility-changes for Rule-based Adaptation of Dynamic Architectures

Sona Ghahremani, Christian M. Adriano, and Holger Giese
Hasso Plattner Institute for Digital Engineering
Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam, Germany
Email: {sona.ghahremani|christian.adriano|holger.giese}@hpi.uni-potsdam.de

***Abstract*—It has been shown for architecture-based self-adaptive systems that rule-based adaptation can be steered by predicting changes on the system utility. However, it can be challenging to build predictions for systems with black-box performance models. The lack of detailed information about the system performance can make it difficult to construct an analytic representation of the system utility. We, therefore, address in this paper how to systematically learn the system utility without relying on detailed knowledge of the system. We developed a methodology that extends the standard machine learning process with regards to validating and selecting prediction models. We evaluated our methodology over a real system over a range of utility complexities and different machine learning methods trained with real failure traces that are publicly available. Our findings are promising in a sense that they suggest that our methodology is applicable to real system failures that could be reproduced on dynamic architectures with different configurations.**

I. Introduction

As introduced in [1,2], rule-based adaptation of architecture-based self-adaptive systems can be steered via predictions of the impact of the rule adaptations on the system utility. For adaptations that can be optimized in a greedy manner, it has been shown that these systems can behave optimally and faster than a system employing a more expensive solution such as a constraint solver [1].

Analytically constructing a system utility is a challenging task. Non-linearities, complex dynamic architectures, and black-box models are a few of the sources of uncertainty and subjectivity. These require specialized domain knowledge [3,4], which makes utility elicitation not trivial [5].

The common practice is to construct the system utility by hand-picking the features of the architecture [6] or reducing the number of the features by exploring inter-feature relationships [7]. In our methodology we side-step this problem by optimistically accepting the selection that is automatically made by certain types of machine learning algorithms, for instance, decision trees [8] and boosting methods [9], [10]. Another common practice is to learn decisions based on the user preferences rather than the system utility [11] directly. However, in the context of self-adaption of dynamic architectures, we cannot expect to have a perfect (or omniscient) decision maker that knows users' preferences at any given time and under different system failure conditions. Our methodology precludes us from having a detailed knowledge of the user preferences, the system, and the specific adaptation decisions necessary to restore the system from a failure. To the best of our knowledge, no attempts have been made so far to learn the system utility for the specific cases of rule-based self-adaptive systems with dynamic architectures.

This paper studies whether we can learn instead of construct the system utility. Our first contribution is to suggest a methodology that extends the standard machine learning process to contemplate the class of self-adaptive systems with dynamic architectures employing rule-based adaptation. The second contribution is to evaluate the suggested methodology concerning the following three research questions: RQ1: Is it possible to systematically learn a prediction model for the utility-changes in a utility-driven rule-based self-adaptive system (particularly when the detailed knowledge of the system is not available)? RQ2: Do the prediction models lead to a system performance that approximates an analytic-defined optimum (real loss lower than 10%), considering that different arbitrary bounds could be chosen to trade between model accuracy and model runtime effort? RQ3: Can the suggested methodology properly *select* the best prediction model without requiring the models to be deployed on a real system?

The paper is structured as follows. The prerequisites in Section II. Section III outlines our methodology. Section IV presents the results of four utility function variants (answering *RQ*1). We evaluate our methodology in section V in two ways. We compare the predicted to the optimal system performance (answering *RQ*2) We compare the system performance of the alternative predictions models to evaluate the ability of the methodology to select the best models (answering *RQ*3). We close the paper with a discussion of the related work (Section VI), conclusion and future work (Section VII).

II. Prerequisites

A. Architecture-based Self-Adaptation

A self-adaptive system is capable of modifying itself at runtime in response to the perception of the environment and the system itself. Violation of certain functional and nonfunctional goals triggers the self-adaptation [12]. Equipping the software system (*adaptable software*) with an external *adaptation engine*, such as a *MAPE-K* feedback loop, enables the realization of self-adaptive capabilities. Elements of the *MAPE-K* feedback loop interact with the system and the environment via a runtime model. A runtime model of a managed system is an architectural representation of the underlying system. The runtime model provides the appropriate level of abstraction and is causally connected to the system [13]. Many approaches

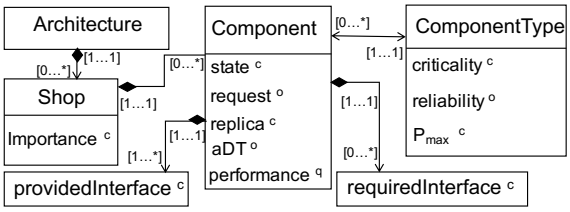

Figure 1: Simplified class diagram of mRUBiS runtime model

consider architecture-based adaptation as a good trade-off between the level of abstraction of the system details and the exposure to the important system-level behaviors and properties [14]. Applying architecture-based self-adaptation implies that all the changes (reconfigurations) are generally achieved by adding, removing, or modifying architectural elements.

The case study employed throughout this paper is mRUBiS [15], an instance of the common RUBiS that is frequently used for validating self-adaptation mechanisms [16]. mRUBiS is an on-line marketplace hosting arbitrary number of shops. Each shop contains 18 components of different types. Components of a shop can adapt and be configured independently of each other. Self-adaptation capabilities are added to mRUBiS via running a *MAPE-K* feedback control loop using the architectural runtime model of mRUBiS. Fig. 1 represents a class diagram of the simplified runtime model. Each concrete runtime model of mRUBiS conform to this class diagram.

The class diagram depicts the basic concepts that construct the configuration space of the architecture (e.g., Architecture, Shop, Component, ComponentType, providedInterface, and requiredInterface as well as the assoications between them). The configuration can be further refined by setting *configuration attributes* such as importance of a shop, state of a component, replica number of a component, or criticality of a component type (all marked with a ^{c}) to the required values. As a simplification, we denote a concrete configuration as a vector of configuration settings $\bar{c}$, and thus the adaptation of a system. i.e., the system is adapted via changing the current configuration $\bar{c}$ to the new configuration $\bar{c}'$.

The *observable attributes* in the model (all marked with an ^{o}) are monitored at runtime (e.g., requests for each component) and reflect the impact of the environment on the system or its context. Similarly, we denote a concrete observation as a vector of observation values $\bar{o}$. In rare cases, the current observations $\bar{o}$ may change to the new observation $\bar{o}'$ independently of the configuration changes. A pair of a configuration and the corresponding observations $(\bar{c}, \bar{o})$ can then be interpreted as the state of the system concerning adaptation [17]. Both $\bar{c}$ and $\bar{o}$ together denote the design space of a self-adaptive system and provide a rich enough language to capture all the principal decisions that are significant for the task at hand [13].

Finally, there are also *quality attributes*[1] such as the performance of each component (marked with a ^{q}) that capture the evaluation of the system operation. Quality attributes usually depend highly on the overall configuration as well as observable attributes. Quality attributes $\bar{q}$ are properties of the system which are orthogonal to its functional goals [18]. Examples of quality attributes are availability, throughput, response time, etc., which may depend on the configuration $\bar{c}$ and observation $\bar{o}$. We denote the concrete quality attribute values as a vector $\bar{q}$. This vector $\bar{q}$ may change to the new quality attribute values $\bar{q}'$ in response to changes on the configuration or observations.

[1]Quality attributes are also often named key performance indicators ($KPIs$) [13] or service-level attributes

B. Utility-driven Rule-based Adaptation

A *rule-based adaptation* can be triggered by a system failure, which causes an adaptation rule r to be selected when the rule r matches its precondition. A match m for the rule r is a fragment of the runtime model that satisfies the precondition of the rule. $RM(\bar{c}, \bar{o})$ is the set of all possible combinations of rule r and match m for a given runtime model with configuration $\bar{c}$ and observations $\bar{o}$. The configuration $\bar{c}'$ comes from applying a possible combination of rule r and matches m for that rule in $RM(\bar{c}, \bar{o})$ and is denoted by $(\bar{c}, \bar{o}) \rightarrow_{r,m} (\bar{c}', \bar{o})$.

The utility function u of a system maps the characteristics of a system to a scalar value. The utility values can represent the level of desirability. In general, for a software system, u can be obtained from Service Level Agreements (SLA's), preference elicitation, or simple naive templates [19]. Utility of a system can usually be defined as a function of system quality attributes $\bar{q}$, i.e., $u(\bar{c}, \bar{q}, \bar{o})$. This function is a quality metric that objectively calculates how well the goals of the system (described by $\bar{q}$ for $\bar{c}$ and $\bar{o}$) are satisfied.

However, for complex systems, it is not trivial to obtain the relevant values for $\bar{c}$, $\bar{q}$, $\bar{o}$ to compute a well-behaving $u(\bar{c}, \bar{q}, \bar{o})$. The reason is that, when planning an adaptation, we often do not know which quality attribute values $\bar{q}$ result from which changes. Instead, we know only the resulting configuration $\bar{c}'$ and the usually stable observations $\bar{o}$. The solution is to predict the quality attribute values $\bar{q}$ by means of a model S that takes as input a planned configuration $\bar{c}$ and observations $\bar{o}$. This way, we obtain the utility function $u(\bar{c}, \bar{q}, \bar{o})$ that can be used to predict the utility of configuration changes in the form of $u^*(\bar{c}, \bar{o}) = u(\bar{c}, S(\bar{c}, \bar{o}), \bar{o})$. In our previous work [1], we analytically derived $u^*(\bar{c}, \bar{o})$ together with the adaptation rules. Fig. 2 illustrates the blue prints of the employed utility-driven self-adaptation.

In the planning step of a given system state $(\bar{c}, \bar{o})$, the utility function calculates the changes in utility (utility-changes) of each $(r, m) \in RM(\bar{c}, \bar{o})$. These utility-changes consist of the differences between the previous and the calculated utility. i.e., $u^*_\Delta(\bar{c}, \bar{o}, r, m) := u^*(\bar{c}', \bar{o}) - u^*(\bar{c}, \bar{o})$, where $\bar{c}'$ comes from $(\bar{c}, \bar{o}) \rightarrow_{r,m} (\bar{c}', \bar{o})$. This approach always chooses the rule r with the maximum $u^*_\Delta(\bar{c}, \bar{o}, r, m)$. The selected (r, m) are then sorted by their utility-increase relative to their execution cost. This way, the (r, m) with higher utilities are applied first to the failing system. Note that instead of $u^*_\Delta(\bar{c}, \bar{o}, r, m)$, a simpler function $u^*_\Delta(\tilde{\bar{c}}, \tilde{\bar{o}}, r, m)$ can usually be employed, because we assume that we do not need the full architectural configuration $\bar{c}$ and observations $\bar{o}$. Instead, we can consider only the subvectors for the architectural configuration $\tilde{\bar{c}}$ and observations $\tilde{\bar{o}}$ which are selected locally and relative to the rule match m.

C. Machine Learning

In a supervised machine learning (ML) approach [20], we can build a prediction model by observing an output y and features $x_1, \ldots, x_n$ and assume a functional relationship $y =$

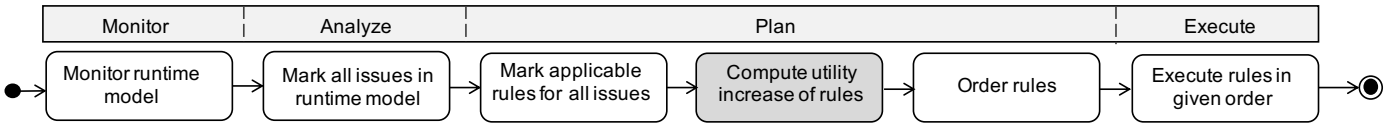

Figure 2: Adaptation steps within a MAPE-K feedback control loop

$f(x_1, \ldots, x_n) + \epsilon$. We can then estimate f using a *prediction model* f^* such that $y^* = f^*(x_1, \ldots, x_n)$ approximates the functional relationship with a *prediction error* denoted by $|y - \epsilon - y^*| = |f(x_1, \ldots, x_n) - y^*| = |f - f^*|$.[2] However, building a prediction model f^* also involves iterations across a set of standard steps (Fig. 3) that we explain next.

Preparing (1) includes three sub-steps. Training and validation data (*data generation* (1.1)), choosing and setup of the machine learning algorithm, e.g., number of trees in a random forest (*method choice and tuning* (1.2)), and selecting a subset of features $x_1, \ldots, x_n$ to consider during training (*feature selection* (1.3)). *Training* (2) runs the machine learning algorithm of choice that builds the model f^* optimized for a training error, e.g., Root Mean Square Error (RMSE). This training error is obtained by testing the prediction model against data not seen during training, e.g., using K-fold cross-validation. *Validating* (3) happens iteratively with steps 1 an 2 for each prediction model f^*. The goal is to *check prediction errors* (3.1) and to *check runtime effort* (3.2). Validation mitigates over-fitting by using datasets that are distinct from training (e.g., 70% for training and 30% for validation). If the validation indicates that the prediction model is not yet appropriate, we can obtain more data, choose a new the method/hyper-parameters, or change feature selection. At some point we deem the prediction model acceptable regarding error and model size/complexity. Since this stopping criterion is subjective, one usually needs to explore multiple prediction models to be confident in the outcome.

III. METHODOLOGY

We address the learning problem with a four step methodology (Fig. 3) that extends (shown as gray*) and modifies (marked with *) the standard machine learning process (section II-C). The extensions include: considering multiple prediction models in parallel, adding a dedicated *selecting* step (4), introducing an additional activity to the *validating* step (3.2), and adjusting some of the activities (1.1, 1.3) to make machine learning applicable to our specific problem.

The goal of the methodology is to find a proper approximation for the analytical utility-change function $u^*_\Delta(\bar{\tilde{c}}, \bar{\tilde{o}}, r, m)$ by means of a prediction model $\hat{u}^*_\Delta(\bar{\tilde{c}}, \bar{\tilde{o}}, r, m)$ that replaces the activity in gray (Fig. 2). Consequently, $u^*_\Delta(\bar{\tilde{c}}, \bar{\tilde{o}}, r, m)$ is the output and $\bar{c}, \bar{o}, r, m$ are the features we observe for the functional relationship that has to be estimated. As we can measure $\bar{q}, \bar{c}, \bar{o}, r, m$ by running the system, we can also indirectly observe $u^*_\Delta(\bar{c}, \bar{o}, r, m) := u^*(\bar{c}', \bar{o'}) - u^*(\bar{c}, \bar{o})$. Note that $u(\bar{c}, \bar{q}, \bar{o})$ corresponds by definition to the unknown $u^*(\bar{c}, \bar{o})$. The outputs (i.e. utility-changes) correspond to the feedback signal for the learning techniques [20]. This is important because it makes the format of the collected data conducive to be used by a supervised machine learning method.

Step 1 Preparing: The *-annotation for the sub-steps represent our new ideas to modify and extend the standard machine learning process introduced in section II-C. In *data generation* (1.1*), we have to mitigate specific realizations of the adaptation engines that can inject bias in the data. *feature selection* (1.3*) is challenging for dynamic software architecture, because the number of configuration and observation attributes can grow arbitrarily. Since features can be composed with various arithmetic operators, analytically discovering an equation involves a search in combinatorial space that can grow fast, even for small architectures.

Step 2 Training: No adjustments needed in our methodology w.r.t. to the standard machine learning process (section II-C).

Step 3 Validating: To compare prediction models across different scenarios, we utilized the Mean Absolute Deviation Percent (MADP), which gives normalized values between zero and 100%, $\text{MADP} = 100 \cdot |\frac{Actual - Predicted}{Actual}|$ In addition to the typical *check prediction errors* (3.1) (section II-C), we also need to *check runtime effort* (3.2*) for the prediction models. The reason is that, the runtime effort of the prediction model also affects the performance of the system. Since we can use the data generated in step 1.1*, we can measure the runtime effort without having to run the real system.

Step 4 Selecting: In the case that we have learned several acceptable prediction models, we need to choose one to be deployed on the adaptation engine. However, model selection is known to be a difficult task [21]. Furthermore, relying on prediction error (MADP) does guarantee the best adaptation decision. Our concern is the ordering and selection of the adaptation rules with highest utility-change, rather than the accuracy of the prediction (value of the changes). Therefore, we have to identify a proper selection procedure that takes this into account.

Finally, applying the methodology determines one prediction model that approximates the optimal utility-changes. The prediction model is then exported as a runtime executable format. This replaces the analytic utility function (gray step in Fig. 2). Next we detail our proposed extensions and modifications to the standard machine learning process.

A. Step 1.1*: Data Generation

In *data generation* (1.1*), we execute the adaptable system together with a randomly operating adaptation engine on top. This mitigates bias in the data that can be introduced by a specific realization of the adaptation engine. A random adaptation engine maps the configuration and observation attributes to the adaptation rules in a random manner. It provides a thorough coverage of the configuration-rule combinations. Conversely, any other (rational) adaptation steering policy could introduce bias by leaving out certain combinations of configuration, observations, and adaptation rule matches.

B. Step 1.2 Method Choice and Tuning

To learn a prediction model $\hat{u}^*_\Delta(\bar{\tilde{c}}, \bar{\tilde{o}}, r, m)$ that approximates $u^*_\Delta(\bar{c}, \bar{o}, r, m)$ in line with best practices in machine learning,

[2]Note that the *real error* (without noise) would be $|y - y^*| = |f + \epsilon - f^*|$.

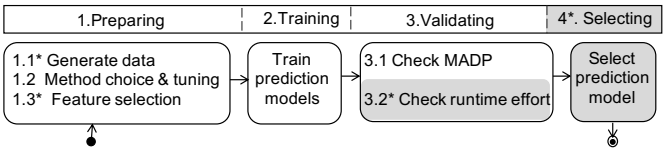

Figure 3: Suggested methodology

we suggest to use methods based on decision trees. These methods automatically perform feature selection and are capable of modeling non-linear functions. Therefore, contrary to the linear regression methods, we do not have to hypothesize which features are used in the utility functions and how they are combined into an analytic formula. However, decision trees pose two major challenges: higher risks of over-fitting and bias. Over-fitting happens when the prediction model presents low error rate for the training data, but high error rate with data that the model has not seen (validation data). This is a consequence of overly complex models (e.g., deep decision trees), which end up factoring noise into the solution. Bias consists of high variability in the predictions across training sessions. Bias might be a consequence of low signal-to-noise relationship or data that is not representative of all functional relationships that we need to learn.

We mitigated these challenges in two ways: ensemble machine learning methods and loss function optimization over their hyper-parameters [22]. For both approaches, we adopted three different learning methods which were extensively used by the practitioners in the machine learning community: Random Forest (RF) [8], Gradient Boosting Regression Models (GBM) [9], and Extreme Boosting Trees (XGB) [10].

C. Step 1.3* Feature Selection

The problem in *feature selection* (1.3*) is that, for the considered case of dynamic software architectures, the number of configuration and observation attributes can grow arbitrarily such that the combinations of $\bar{c}$ and $\bar{o}$ are not even bounded. However, for the considered case, the learning problem is only to train a prediction model $\tilde{\hat{u}}^*_\Delta(\tilde{\bar{c}}, \tilde{\bar{o}}, r, m)$ that properly approximates $u^*_\Delta(\bar{c}, \bar{o}, r, m)$ for a specific adaptation rule match m. By only considering small architectural fragments (i.e., $\tilde{\bar{c}}$ and $\tilde{\bar{o}}$) and extending the match m of the adaptation rule r, the learning problem can be limited to a smaller fixed number of features and standard machine learning methods become applicable.

The sub-vectors $\tilde{\bar{c}}$ and $\tilde{\bar{o}}$, as the selected features to learn $\tilde{\hat{u}}^*_\Delta(\tilde{\bar{c}}, \tilde{\bar{o}}, r, m)$, are determined relative to the match m for rule r. Therefore, by systematically extending the match of an adaptation rule, such that more features are available, we can generate enough data to train a prediction model (even for large and unknown dynamic architectures). Take for instance when the providedInterface of a component throws exceptions (Fig. 1). The state of the component changes, which makes it possible to match m for a repair rule r by only including the providedInterface and the state attributes. We can systematically extend m to include more features (e.g., attributes of the shop containing the component or requiredInterface).

Here, we only consider how to extend the offered set of features with additional candidates. The reason is that we will keep only the prediction models that themselves select a suitable subset of the offered features (Section III-B). We do this by relying on the training step to reduce the features to the most significant ones.

Note that a constructed analytic function $u^*_\Delta(\bar{c}, \bar{o}, r, m)$ will usually only consider a small fragment of the architectural configuration $\bar{c}$ and observations $\bar{o}$ denoted by $\tilde{\bar{c}}$ and $\tilde{\bar{o}}$, which contain only a few dimensions of the original vectors. Therefore, we assume that a local (hence small) fraction of the configuration and observation attributes correspond to a feature space that is sufficient to learn the utility-changes. Note also that we aim to predict only the impact of each adaptation rule application for a given context, rather than learning the complete model S, which would not be feasible.[3]

D. Step 3.2* Check Runtime Effort

In addition to the standard machine learning validation step, i.e., *check prediction errors* (3.1), we suggest an additional step, *check runtime effort* (3.2*), to validate the k prediction models $\hat{u}^*_{\Delta,1}(\tilde{\bar{c}}, \tilde{\bar{o}}, r, m), \ldots, \hat{u}^*_{\Delta,k}(\tilde{\bar{c}}, \tilde{\bar{o}}, r, m)$.

We suggest to use the data generated for learning to check the runtime effort of the prediction models. Based on the observed execution time and the characteristics of the system, a prediction model might be skipped or further tuned to prevent performance reduction at runtime.

E. Step 4* Selecting

Selecting models by their prediction error does not guarantee that the best reward will be obtained. In other words, minimizing the prediction error does not necessarily maximize the system performance. The reason is twofold. The prediction errors are averages across predictions, hence the error is not necessarily uniform across component types. The second reason is that system performance depends on how the adaptation decisions are applied. This follows a three step process (Fig. 2): (1) estimate the utility-change of the adaptation rules given the current configuration and observation, (2) select an adaptation rule r for a given match m and (3) for all the matches, sort the selected rules by their utility-change. Step (2) and (3) are necessary to maximize the system utility. For each sorted list of adaptation decisions, the adaptations with higher utility-change should be applied first. Nonetheless, even small prediction errors can make the order of adaptations diverge from the optimum.

Our solution is to directly compare the decision lists produced by the prediction models (predicted lists) to optimal lists. The predicted lists can be obtained by executing the prediction models on the real system. For each predicted list, it is also possible to *observe* the real utility-change via measurable quality attributes. Decisions (adaptation rules) are then manually sorted in a descending manner regarding the utility-changes. This generates an optimal list for each predicted lists, which can now be compared. We compared the lists regarding differences

[3]Learning directly the model S can be a first naive solution to (*RQ*1). Model S predicts the quality attribute values $\bar{q}$ for a planned configuration $\bar{c}$ and observations $\bar{o}$. However, this would require a thorough coverage of all combinations of $\bar{c}$ and $\bar{o}$. This can lead to combinatorial explosion regarding the size of the architecture, combinations of $\bar{c}$ and $\bar{o}$ attributes and their domain sizes. Standard machine learning approaches that require a fixed number of features are not directly applicable due to this effect.

(mismatches). For that we extended a set of standard similarity metrics to cover three families of mismatches: the number of items with different ranking positions (counting), the magnitude of the ranking differences (distance), and the location of these differences (position), which we explain next.

Count-based metric - We adopted the Jaccard index [23], that measures the ratio between the intersection and the union of two lists. We also evaluated the Ochiai coefficient [24], but since the *predicted list* is fully contained in the *optimal list*, the denominator of the Ochiai formula becomes the square of the union times the intersection. This causes the Ochiai coefficient to overestimate the similarity between two lists. For this reason we adopted the Jaccard index.

Distance-based metric - Simply counting might hide the fact that mismatches can be distinct regarding the distance between the actual position in the ranking and the predicted one. Distances can be seen as the magnitude of the error for each mismatch. Items with larger mismatch distances would have a larger negative impact on the similarity index. This is important when certain list items should present smaller mismatch distances than others, e.g., more critical system components. A distance-based similarity metric could help us prioritize these components. Non-parametric correlation metrics (e.g., Kendall-tau, Spearman) are alternatives to compute distances among the mismatches. We chose Kendall-tau, because Spearman is known to present worse approximation of the measured population. [25]. However, Kendall-tau does not compare items at the same position. It compares $list_A[i] == list_B[i+1]$, which makes Kendall-tau correlation less strict than the Jaccard metric.

Position-based metric - Mismatches can also be distinct regarding the position that they happened in the ranking. Take for instance an item that was ranked at second position, but should have been ranked first. Now take the case of an item ranked 46th, but should have been ranked 45th. Note that in both cases, there are two mismatches (hence, same counting) and the mismatch distance is the same (exactly one). However, the mismatch at the top might be more relevant. Among several metrics from the information retrieval field [26], we chose one of the simplest ones, the Discounted Cumulative Gain (DCG) metric [27]. Its simplicity allowed us to extend the metric to consider the position of the mismatch and to apply the discount factor only to the item that caused a mismatch.

Aggregation of Metrics - Each similarity metric might reflect an architectural concern, which a designer of an adaptive system would need to combine in different ways. We enabled this by a single metric (similarity aggregation metric - SAM). This metric is normalized between 0 and 1 and still allows a designer to determine distinct levels of importance per metric. Since we did not have any priority about how to rank component types, we set all factors to one and computed the SAM value across all prediction models mod.

$$\text{SAM}(mod) = \frac{\alpha \cdot Kendall(mod) + \beta \cdot DCG(mod) + \gamma \cdot Jaccard(mod)}{\alpha + \beta + \gamma} \quad (1)$$

IV. Application

In the following we present an instantiation of the methodology for mRUBiS, the adaptive system uses as case study.

Table I: CHARACTERISTICS OF FAILURE TRACES

	Failure group size	**IAT(s)**	**Used in**
Grid5000	$LOGN(1.88, 1.25)$	$LOGN(-1.39, 1.03)$	Step 1- 3
LRI	$LOGN(1.32, 0.77)$	$LOGN(-1.46, 1.28)$	Step 4
DEUG	$LOGN(2.15, 0.70)$	$LOGN(-2.28, 1.35)$	Evaluation

Table II: VARIANTS OF mRUBiS

mRUBiS Variant	**Description of the Encoded u^***
Linear	u^* only includes linear elements.
Saturating	There are saturation effects considered in u^*.
Discontinuous	u^* includes discontinuous but linear steps.
Combined	u^* combines all three cases above.

A. Step 1: Data Generation

We equipped mRUBiS with both self-healing and self-optimizing capabilities following the scheme of [1]. The employed adaptation rules are either repair rules (Restart, LightWeightRedeployment, HeavyWeightRedeployment and Replace) or optimization rules (AddReplica and RemoveReplica). Self-adaptation of mRUBiS is triggered by injecting failures to components. Failures (issues in Fig. 2) were obtained from real traces of real computer systems that are publicly available[4]. This guarantees a realistic scenario of self-adaptation.

To generate data for machine learning we employed a failure trace generated from *Grid5000* [28]. These traces include time and space correlated failures. Failures arrive in groups (bursts) and not individually. Table I describes the distributions proposed by [28] for Inter-Arrival Time (IAT) between the failure bursts and the number of the failures at each burst (failure group size). Table I also shows where we used each failure trace. This was important to guarantee independent outcomes among the steps of model building (step 1-3), model selection (step 4) and model evaluation.

To have a good coverage of the configuration space, we equally distribute the failures among components of all shops. Each failure causes a drop in the utility. The random adaptation engine randomly assigns each reached configuration and observation to an applicable adaptation rule r. Applying an adaptation rule resolves the corresponding failure and increases the utility. The overall utility for an instantiation of the mRUBiS architecture (Fig. 1) is the sum of the utility of the shops. The utility of a shop is the sum of the utility of its components. The utility-changes are *observed* via measurable quality attributes (section III-A). that we can measure (observe) independent of the model $S(\bar{c}, \bar{o})$.

We execute *four* variants of mRUBiS. Each operating with a different utility function varying in complexity [29]. Table II summarizes the descriptions of the utility function u^* encoded in each variant of mRUBiS. These functions are only employed on mRUBiS as a *black-box* model to provide the *feedback* for machine learning ([30]). The details of the u^* functions, providing the utility-changes, are hidden to the machine learning method. Finally, we generate the data by executing each variant of mRUBiS with a sample configuration and observation along with the selected rule r and the utility-change.

[4]http://fta.scem.uws.edu.au/ - accessed 14 April 2018

B. Iterate Step 2, 3, and 1: Training, Validating, and Preparing

This subsection answers *RQ*1 for a spectrum of utility function complexities, machine learning methods, and dataset sizes ($1,000$ $3,000$, and $9,000$ data points). We saved 30% of each of these datasets for validation and used remaining 70% for training and testing (10-fold cross-validation).

We enable model validation by enforcing a usual data splits between training and validation. The split with lowest error corresponded to 70% for training/testing and 30% for validation. We also investigated data splits of 80/20 and 90/10 (see report [31]), which did not show better results for the larger datasets. We trained and validated the prediction models using 10-fold cross-validation over a range of hyper-parameter values. Among different prediction models, we chose the ones with the smallest root mean square error (RMSE). The hyper-parameters that presented larger impact on RMSE were the number of trees, the number of elements on the leaf nodes, and the maximum depth of the tree. See report [31] for a detailed results of this tuning process.

To validate the prediction models, we computed the error between the actual and the predicted utility, defined for our case as: $e^*_\Delta(\bar{c},\bar{o},r,m) := |u^*_\Delta(\bar{c},\bar{o},r,m) - \hat{u}^*_\Delta(\tilde{\bar{c}},\tilde{\bar{o}},r,m)|$. This error was further normalized (as MADP) to be compared across different datasets. We minimized MADP by allowing the decision trees to select all the available input features. We accomplished this by tuning various hyper-parameters such as number of nodes in the leaves and maximum depth of trees. This was confirmed in the models exported to predictive model markup language format (pmml)[5].

The validation results for the Combined variant across the three prediction models showed that larger dataset sizes corresponded to lower MADP values (Fig. 4). The same pattern was observed for the other three utility functions. This suggests that we can optimize for accuracy by adopting the larger dataset ($9K$). We also investigated larger datasets ($10K$, $100K$), but the prediction error saturated after $10K$.

All three prediction models, presented increasing MADP as the complexity of the utility function increased (Fig. 5). Except for the Discontinuous variant, GBM and RF models presented MADP values that are less than 0.5% different. The MADP value for the XGB was twice to three times smaller than the ones from the GBM and RF. This suggests that the XGB model would probably perform better when deployed on the running system. However, it is not clear whether GBM or RF would perform better for all the utility functions.

We were however surprised with the results for the Discontinuous utility. It required at least 3,000 data points to reach the same MADP level of the Saturating and Linear utilities with 1,000 data points. We initially expected that the Discontinuous utility would present lower prediction error than the Saturating utility. Our intuition was that the Discontinuous utility, which is still linear within the continuous intervals, would be easier to be captured by the decision tree methods than the Saturating utility, which is non-linear. Further investigation is required to understand why the discontinuities were not precisely identified in the final decision trees.

[5]http://dmg.org/pmml/v4-0-1/GeneralStructure.html accessed 12 April 2018

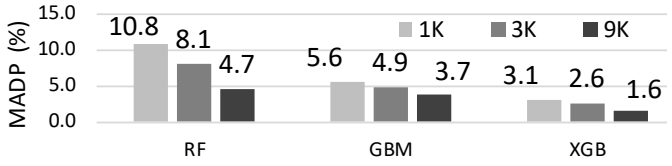

Figure 4: MADP for the Combined utility function across prediction models trained with all dataset sizes

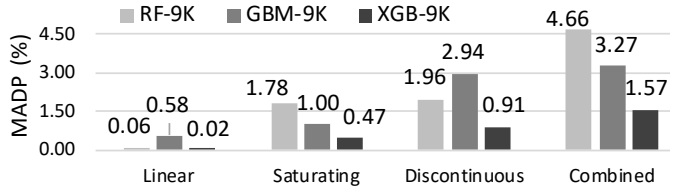

Figure 5: MADP across utility functions and prediction models trained with 9K datasets

To check the runtime effort of the prediction models, we used 9K datasets for training (Section III-D). While automatic feature selection and hyper-parameter tuning allowed us to minimize prediction error, it could also increase the size of the decision trees. This is critical because the larger the decision tree, the longer it takes to make runtime predictions. Therefore, we tuned the models to keep runtime effort within one order of magnitude between each other. Take for instance the RF model that showed the highest prediction error. We further minimized MADP for this model (now RF-Heavy) by respectively reducing the minimal number of elements in the leaves (10 to 5) and increasing the number of trees (100 to 200). While RF-Heavy presented a reduction in MADP from 4.66% to 4.09% for the Combined utility function, the runtime effort increased in more than two orders of magnitude (from 67,065 μs to 8,723,761 μs). For this reason, we chose the model with slightly larger MADP. We performed this procedure for all the other models (Fig. 6).

The choice of learning method presented a larger impact on runtime effort than the choice of utility function complexity. Since distinct utility complexities imply different sets of features, one possible explanation for the results in Fig.6 is that, the number of features has a smaller impact on model size than the hyper-parameter tuning.

Answering *RQ*1: We confirmed that (1) larger dataset sizes provide lower MAPD values for all prediction models and (2) prediction models can be tuned to minimize prediction error and runtime effort.

C. Step 4 :Select the Prediction Model

In this section we select the prediction models based on their produced adaptation decisions as outlined in section III-E. We

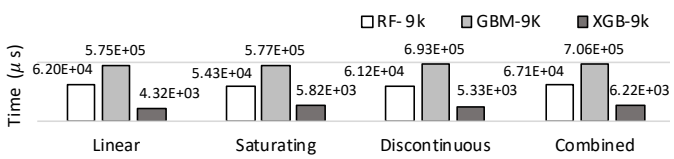

Figure 6: Runtime effort across functions in log scale

Table III: SAM VALUES FOR PREDICTION MODELS (9K)

Variant\Prediction Model	**RF**	**GBM**	**XGB**	**Selected**
Linear	**0.959**	0.521	0.871	RF
Saturating	0.742	0.770	**0.812**	XGB
Discontinuous	**0.891**	0.864	0.842	RF
Combined	0.809	**0.812**	0.731	GBM

selected the prediction models based on the similarity between the optimal and the predicted list. The different predicted lists were produced by executing the prediction models on the mRUBiS variants with 50 failure cycles randomly sampled from the LRI failure trace. We did the following to generate the optimal list for each predicted list. We injected the exact same samples of the LRI trace to mRUBiS and observed the utility-changes (u^*_Δ). Then we manually ordered the list in a descending manner regarding the utility-changes.

The similarity measures presented three distinct patterns (Fig.7): proportionate low values for GBM-*linear* function, less discriminative values for Kendall-tau, and the same relative values for the DCG and Jaccard metrics.

The much lower values for GBM-*linear* utility can be explained by the large difference in prediction error (MADP) between GBM versus RF and XGB models (Fig.5). GBM model has MADP respectively almost 10 to 30 times larger than of the RF and XGB models. This shows that a prediction error considered small (0.58%) can have a large impact on the quality of the adaptation decisions.

The reason of this high impact is the simplicity of the Linear utility function, which produces utility-change values with a lower variance if compared to the more complex functions. Hence, even small errors are sufficient to alter the order of the adaptation decisions. This might suggest a surprising trade-off. While simpler utilities are easier to learn (i.e., small prediction error), the adaptation mechanisms deployed with these simpler utilities are orders of magnitude more sensitive to error variance in these models.

Kendall-tau values did not discriminate the prediction models as well as the DCG and Jaccard metrics. Except the Linear utility, for all the other utility complexities, the similarity differences for Kendall-tau are close to 1% (Fig.7). Conversely, the DCG and Jaccard values range from at least 2% to 15%. Although these differences are not the same, DCG and Jaccard ranked the prediction models in the same way.

Ultimately, after computing the SAM values (Eq. 1) across all models, we were able to identify the best performing model (Table.III).

V. EVALUATION

We evaluate the quality of the results to answer the two remaining research questions. The quality is relative to an optimal and a baseline benchmark. We report on the accuracy of the criteria used for model selection. To guarantee a fair comparison, we detailed the data and system evaluation setup.

A. Ground Truth and Evaluation Setup

The existence of the ground truth (an optimal steering strategy) is usually not feasible for systems that come with a black-box performance model [3]. Conversely, with a simulator, we can access all configurations and undo rule executions.

Comparison: A simulator at first enables us to deploy multiple prediction models under the same conditions and therefore compare their performance in a fair manner. Secondly, in a simulator, we can observe the utility-changes that result from applying a rule (ground truth) and undo the application step. This allows us to determine the *best* and the *worst* rule application and emulate accordingly the best possible strategy (Optimal) as well as the worst possible one (Baseline). Overall, we thus can consider for the comparison, the Baseline, the Optimal strategy, and the considered prediction models RF, GBM, and XGB. To facilitate model comparison, we normalized the reward values between zero and one as follows:

$$\text{Normalized Reward}\,(mod) = \frac{\text{Reward}\,(mod) - \text{Reward}\,(\text{Baseline})}{\text{Reward}\,(\text{Optimal}) - \text{Reward}\,(\text{Baseline})}. \quad (2)$$

Evaluation Setup: We generated the ground truth by running four simulations of mRUBiS (100 shops each). One simulation for each variant in Table II. With the same setup, we also determined the optimal strategy (Optimal) and the baseline (Baseline) that will serve as benchmark to evaluate the prediction models built and selected by the methodology. To mitigate bias in the evaluation, we injected the simulator variants with a real world failure trace (DEUG in Table I) that was not used in the steps 1-4 (Fig. 3). For each variant, we injected 50 bursts of failures that resulted in 50 executions of the *MAPE-K* loop. The number of the failures at each burst and the inter-arrival time followed the distributions in Table I.

Variants: While Table II describes the encoded utility function u^* in each variant, Table IV presents the relevant fragment of u^* for each utility-change (u^*_Δ) to be estimated by the prediction models.

Linear variant: Following the scheme of [1], we equip one of the mRUBiS simulators with a linear function with feature interaction [32], i.e., a polynomial of degree one.

Saturating variant: We extend the linear variant by introducing the *quality attribute* performance to the formula as presented in Table IV. The performance of a component is formulated as $\text{Performance} = \text{P}_{max}\ \tanh(\alpha\frac{\text{replica}}{\text{request}})$, where α defines the gradient of the hyperbolic tangent function for each componentType. Using a hyperbolic tangent function to map the $\bar{c}$ and $\bar{o}$ attributes to performance provides a saturating effect[6]. Each componentType has a unique P_{max} (to which the performance of the component saturates). Also a particular ratio of $\frac{\text{replica}}{\text{request}}$ defines the shape of the function. Therefore, for each componentType the function has a different shape. Any change in the number of the requests either results in reduced performance (the number of requests has increased) or results in saturating performance (the number of requests has decreased). Both situations are considered non-optimal and trigger the optimization rules AddReplica and RemoveReplica.

Discontinuous variant: We extended the *linear* variant generating discontinuity based on providedInterface (pI) attribute. pI of a component z in mRUBiS architecture describes to which degree other components depend on z. Components

[6]In saturating functions the initial stage of growth is approximately exponential. As the saturation begins, the growth slows down and eventually stops at maturity.

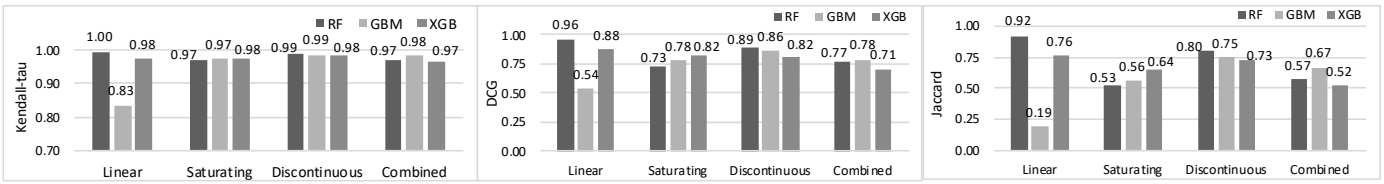

Figure 7: Similarity aggregation values for the optimal and the predicted decisions produced by the selected prediction models

Table IV: GROUND TRUTH FOR SIMULATOR VARIANTS

Fragment of u^* relevant for u^*_Δ	**Variant**
Reliability Criticality (providedInterface + requiredInterface)	Linear
Reliability Criticality $\mathsf{P}_{max}\ \tanh(\alpha\frac{\text{replica}}{\text{request}})$ (providedInterface + requiredInterface)	Saturating
Reliability Criticality (requiredInterface+1) Importance β(providedInterface) $-$ 10ADT	Discontinuous
Reliability Criticality Importance β(providedInterface) P_{max} $\tanh(\alpha\frac{\text{replica}}{\text{request}})$ (requiredInterface+1) $-$10ADT	Combined

with larger pI have a larger weight in the utility function by defining $\beta = \mathsf{pI}$ if $2 \leq \mathsf{pI}$; otherwise, $\beta = \mathsf{pI}$+1. Note that the importance attribute in the formula is a shop-level attribute and not a component-level attribute (Fig. 1). Therefore, the *discontinuous* variant extends its domain to also include the attributes of the shop. ADT (Average Deployment Time) of a component introduces a notion of cost to the utility function by adding the time attribute.

Combined variant: For this variant, we accumulate the previously introduced features in one function. Note that for all the variants, the prediction models are used to estimate the changes in the utility as a result of rule application ($u_\Delta{}^*$) and not the utility function itself (u^*).

B. Evaluating the prediction model performance - RQ2

To evaluate prediction model performance, we measured the normalized reward of the adaptations applied to the simulator variants and the runtime effort of the models. We calculated the normalized reward for two scalability scenarios: different datasets sizes and mRUBiS architectures sizes.

The normalized reward increased for all prediction models trained with larger *dataset sizes*. Fig. 8 shows results for the Combined variant. The same pattern was observed for the other three prediction models across all four variants. This confirms our earlier finding for the prediction error (Fig. 4). It also suggests that deciding for larger datasets (9k) was a correct decision. The worst reward loss value was 5.5% for the RF model (9K) that predicts the Combined utility (Fig. 9).

As the *architecture size* grows, the normalized reward did not vary above 10% (Table V). We observed similar patterns for the RF and GBM models for dataset size $9K$. To show this,

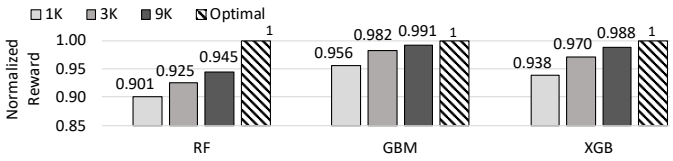

Figure 8: Normalized reward across prediction models for the combined variant computed with the DEUG trace

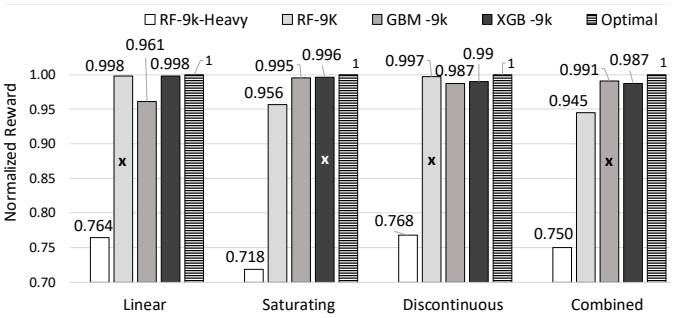

Figure 9: Normalized reward across prediction models

we deployed the prediction models on mRUBiS simulators with different architecture sizes (100, 500, and 1000 shops). In addition to the DEUG trace, we injected the simulator variants with LRI trace. These results confirmed that the performance of the prediction models scales with architecture size. This also might suggest that we could transfer prediction models learned in smaller architectures to larger ones. This of course would require that the range of configuration and observation attributes (features) remain the same across architecture sizes.

Runtime effort scaled for the prediction models across all simulator variants. We evaluated this by measuring 20,000 executions of each prediction model in each simulator variant.[7] The standard deviations of the outcome of each of these executions were below 0.5%. This confirms that the findings presented in Fig. 6 are accurate.

Answering *RQ*2: For the final prediction models trained with 9K datasets, the worst reward loss was 5.5%, i.e., a considerably smaller error than the 10% error bound.

C. Evaluating the prediction model selection - RQ3

Prediction models with higher similarity metric values also presented higher normalized reward (Fig. 9). We also marked with an "**X**" the prediction models that were selected based on the SAM values (Table.III). Comparing the normalized reward across all models in Fig. 9, we can confirm that the similarity metrics selected the best performing models within each variant (i.e., largest normalized rewards).

Although the RF-Heavy model presented better prediction error (MADP) than the other RF model, the RF-Heavy model presented more than 10% loss in reward across all variants (Fig. 9). This confirms that during model tuning (step 1.2) and validation (step 3), it was sensible to trade slightly higher error rate for a smaller runtime effort.

[7]The experiments have been conducted on a machine with OS X 10.13, Intell processor 2.6 GHz core i5, and 8 GB of memory

The interesting observation is that the prediction model with the smallest runtime effort (XGB) did not achieve the highest normalized reward (Fig. 9). One possible explanation could be the effect of the different failure trace distributions (Table. I). The average of inter-arrival time between the group of failures in DEUG trace (i.e. the frequency of self-adaptation), is 1543.26 seconds. To execute an adaptation that resolves all the failures injected in one cycle, the slowest prediction model (i.e., GBM model for the Combined variant in Fig. 6) takes on average 38.18 seconds. This means that, even for the slowest model, the overhead for each adaptation is on average 38.18 seconds. Therefore, given the inter-arrival time of 1543.26 seconds, this overhead does not have a long lasting impact on the reward. However, the impact of the overhead can be different in the presence of failure traces with different distributions.

Answering *RQ*3: The best performing prediction models in Fig. 9 are the same as the ones suggested by the aggregated similarity metrics (SAM Table III). The selected models presented reward loss of less than 1% (relative to the optimal reward). This confirms that our methodology, in (step-4), properly *selected* the best prediction models. Moreover, each of the lower performing models were correctly ranked, as they presented the same ranking positions by SAM values (III) as by normalized reward (9). Ultimately, this was possible because during the model tuning iterations (steps 2, 3, and 1) we did not discard any prediction model that could have potentially outperformed the alternatives (section IV-B).

D. Threats to Validity

Internal validity - The similarity metrics can be too optimistic, because we do not count as a mismatch the cases of components with the same utility-change (i.e., same ranking position). However, this happened only for the Linear utility function, which already presented low prediction errors. *Conclusion validity* - Slightly different failure traces can cause completely different adaptation outcomes. Therefore, we carefully generated and tested failure traces to be independent and identically distributed across the different ranges of dataset sizes and utility functions. This involved testing the generated traces to detect inconsistent output from the simulator, e.g., zero utility value. Inconsistencies are part of the inherent randomness of a simulation of a real system. *External validity* - Three factors hinder the generalization of our findings: size/quality of the failure traces, the specification of the components/shops, and the available adaptation rules. For a different setup of these factors, the current prediction errors might be unacceptable (even when small). This might happen in the case that adaptation decisions must present variances in utility that are smaller than the prediction errors.

Table V: NORMALIZED REWARD OF XGB-9K FOR DIFFERENT ARCHITECTURE SIZES ACROSS VARIANTS

Variant	Failure trace DEUG			Failure trace LRI		
	100	500	1000	100	500	1000
Linear	0.998	0.997	0.997	0.997	0.997	0.997
Saturating	0.996	0.997	0.995	0.996	0.996	0.996
Discontinuous	0.990	0.990	0.989	0.992	0.992	0.991
Combined	0.987	0.987	0.986	0.990	0.991	0.990

VI. Related Work

Runtime planning heavily depends on an accurate performance reasoning. Model-based performance prediction techniques have been vastly practiced [33]. Time series techniques [12] are applied for predictive modeling of response time. Machine learning methods have been used to model on-line QoS for cloud-based software services [34] and virtualized applications [35]. [7] proposes a generic feature-oriented methodology to reduce the dimensions of the configuration space. Although we do not propose a solution for feature selection, we avoided this problem by allowing the decision trees to perform it automatically. Performance prediction and reasoning have also been used extensively investigated in the communities of control theory [36]. [37] employs reinforcement learning for on-line planing. [38] proposes an on-line, unsupervised learning as a model-free solution to explore alternative system assemblies. [39] uses on-line learning to steer adaptation at runtime. We opted for off-line learning which allowed us to investigate problems of scalability and cold-start (not enough data for training). [40] uses machine learning to cope with dynamics that cannot be captured by the analytical model as a result of migrating from the simulator to the real system. [41] proposes a hybrid approach that combines queuing network with reinforced learning to make resource allocation decisions. Note that in multi-agent domain the state space or configuration space is assumed to be fully discoverable while in the field of self-adaptive systems this space can be extremely large and as pointed by [42], the computational cost of the search for an optimal architectural configuration increases exponentially. We tackled this problem by systematically covering the configuration space via running a random simulator to generate data and using complex functions with high variability as ground truth. Other automated planning techniques (i.e., reinforcement learning [37] and genetic algorithms [43]) have been explored to generate adaptation plans at run time. [6] defines a cost model to transform the traditional view of model learning into a multi-objective problem that considers model accuracy and measurements effort. Similarly, we are also trying to learn the prediction models for self-adaptive systems, but in addition, we also propose a methodology that generates and selects models without the need to run a real system. Ultimately, in preference learning [11] the goal is to learn a utility function when the user preference information is uncertain. We instead do not require the knowledge of the user preferences or the system internal performance model to learn the system utility.

VII. Conclusion

In this paper we investigated if it is possible of learning instead of constructing the system utility that steers rule-based adaptation in self-adaptive systems with dynamic architectures. Our approach was to design a methodology that extends the standard machine learning process. Our results were promising and can be summarized in the answers to three research questions. *RQ*1 - it was possible to systematically learn a prediction model for the utility-changes in a utility-driven rule-based self-adaptive system, even when the detailed knowledge of the system was not available. *RQ*2 - the obtained prediction models led to a performance that was as good as the optimal. The maximal loss of performance compared with the optimal solution was 5.5%, which is considerably lower than the 10% upper-bound used as a stopping criterion in our methodology.

This suggests that it might still be possible to obtain good performing models by accepting models with prediction errors larger than 10%. We also showed that the prediction models scaled for larger architectures and over multiple *MAPE-K* loops. *RQ*3 - our methodology always selected the prediction model with the best performance. Moreover, without the information about the final system performance, our methodology was able to correctly rank all alternative prediction models. While our evaluation was limited to one architecture, we offered a general methodology and evaluated it with real failure traces under different scalability scenarios. We hope this can help other researchers to reproduce our findings on other self-adaptive dynamic architectures. As future work, we plan to integrate the utility prediction with the rule-based decision models. One interesting approach is to use the decision mismatches as part of an expanded machine learning process. For that we envisage a closed loop between making decisions and predictions, so that we can adapt the former as a means of training the latter.

REFERENCES

[1] S. Ghahremani, H. Giese, and T. Vogel, "Efficient Utility-Driven Self-Healing Employing Adaptation Rules for Large Dynamic Architectures," in *ICAC*, 2017, pp. 59–68.

[2] ——, "Towards Linking Adaptation Rules to the Utility Function for Dynamic Architectures," in *SASO*, 2016, pp. 142–143.

[3] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence Models for Highly Configurable Systems," in *Joint Meeting on Foundations of Software Engineering*, 2015, pp. 284–294.

[4] R. Patrascu, C. Boutilier, R. Das, J. O. Kephart, G. Tesauro, and W. E. Walsh, "New approaches to optimization and utility elicitation in autonomic computing," in *AAAI*, 2005.

[5] C. Boutilier, R. Das, J. O. Kephart, G. Tesauro, and W. E. Walsh, "Cooperative negotiation in autonomic systems using incremental utility elicitation," in *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, ser. UAI'03, 2003, pp. 89–97.

[6] P. Jamshidi, M. Velez, C. Kästner, N. Siegmund, and P. Kawthekar, "Transfer learning for improving model predictions in highly configurable software," in *SEAMS*, 2017, pp. 31–41.

[7] N. Esfahani, A. Elkhodary, and S. Malek, "A learning-based framework for engineering feature-oriented self-adaptive software systems," *IEEE transactions on software engineering*, vol. 39, 2013.

[8] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[9] J. H. Friedman, "Stochastic gradient boosting," *Computational Statistics & Data Analysis*, vol. 38, no. 4, pp. 367–378, 2002.

[10] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *ACM SIGKDD*, 2016, pp. 785–794.

[11] T. D. Nielsen and F. V. Jensen, "Learning a decision maker's utility function from (possibly) inconsistent behavior," *Artificial Intelligence*, vol. 160, no. 1-2, pp. 53–78, 2004.

[12] B. Schmerl, J. Andersson, T. Vogel, M. B. Cohen, C. M. F. Rubira, Y. Brun, A. Gorla, F. Zambonelli, and L. Baresi, "Challenges in Composing and Decomposing Assurances for Self-Adaptive Systems," in *SEfSAS III: Assurances*, R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese, Eds. Springer, 2017.

[13] N. Bencomo, R. France, B. H. Cheng, and U. Assmann, Eds., *Models@run.time*, ser. LNCS. Springer, 2014, vol. 8378.

[14] T. Vogel, S. Neumann, S. Hildebrandt, H. Giese, and B. Becker, "Model-driven architectural monitoring and adaptation for autonomic systems," in *ICAC*, 2009, pp. 67–68.

[15] T. Vogel, "mrubis: An exemplar for model-based architectural self-healing and self-optimization," in *SEAMS*, 2018.

[16] T. Patikirikorala, A. Colman, J. Han, and L. Wang, "A systematic survey on the design of self-adaptive software systems using control engineering approaches," in *SEAMS*, 2012, pp. 33–42.

[17] Y. Brun, R. Desmarais, K. Geihs, M. Litoiu, A. Lopes, M. Shaw, and M. Smit, *A Design Space for Self-Adaptive Systems*. Springer, 2013, pp. 33–50.

[18] S.-W. Cheng and D. Garlan, "Stitch: A language for architecture-based self-adaptation," *J. Syst. Softw.*, vol. 85, no. 12, 2012.

[19] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, "Utility functions in autonomic systems," in *ICAC*, 2004, pp. 70–77.

[20] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Pearson Education, 2003.

[21] M. Kuhn and K. Johnson, *Applied predictive modeling*. Springer, 2013.

[22] F. Harrell, "Regression modeling strategies," *as implemented in R package 'rms' version*, vol. 3, no. 3, 2013.

[23] P. Jaccard, "The distribution of the flora in the alpine zone." *New phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

[24] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*. IEEE, 2007, pp. 89–98.

[25] D. C. Howell, *Statistical methods for psychology*. Cengage Learning, 2012.

[26] O. Chapelle, D. Metlzer, Y. Zhang, and P. Grinspan, "Expected reciprocal rank for graded relevance," in *CIKM 2009*. ACM, pp. 621–630.

[27] K. Järvelin and J. Kekäläinen, "Cumulated gain-based evaluation of ir techniques," *ACM TOIS*, vol. 20, no. 4, pp. 422–446, 2002.

[28] M. Gallet, N. Yigitbasi, B. Javadi, D. Kondo, A. Iosup, and D. Epema, *A Model for Space-Correlated Failures in Large-Scale Distributed Systems*. Springer, 2010, pp. 88–100.

[29] S. Tomforde, A. Brameshuber, J. Hahner, and C. Muller-Schloer, "Restricted on-line learning in real-world systems," in *CEC 2011*. [Online]. Available: https://doi.org/10.1109/CEC.2011.5949810

[30] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed., 2009.

[31] C. M. Adriano, S. Gharhremani, and H. Giese, "Technical Report - Learning Utility Changes for Rule-Based Adaptation," 2018, https://github.com/christianadriano/ML_SelfHealingUtility/blob/master/TechnicalReport.pdf.

[32] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*. Springer series in statistics New York, 2001, vol. 1.

[33] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: A survey," *IEEE Trans. Softw. Eng.*, vol. 30, no. 5, pp. 295–310, 2004.

[34] T. Chen and R. Bahsoon, "Self-adaptive and online qos modeling for cloud-based software services," *IEEE Transactions on Software Engineering*, vol. 43, no. 5, pp. 453–475, 2017.

[35] S. Kundu, R. Rangaswami, A. Gulati, M. Zhao, and K. Dutta, "Modeling virtualized applications using machine learning techniques," in *ACM Sigplan Notices*, vol. 47, no. 7. ACM, 2012, pp. 3–14.

[36] A. Filieri, H. Hoffmann, and M. Maggio, "Automated multi-objective control for self-adaptive software design," in *ESEC/FSE*, 2015.

[37] D. Kim and S. Park, "Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software," in *SEAMS*, 2009, pp. 76–85.

[38] R. R. Filho and B. Porter, "Defining emergent software using continuous self-assembly, perception, and learning," *TAAS*, vol. 12, no. 3, 2017.

[39] B. Porter and R. R. Filho, "Losing Control: The Case for Emergent Software Systems Using Autonomous Assembly, Perception, and Learning," in *SASO*, 2016, pp. 40–49.

[40] K. Angelopoulos, A. V. Papadopoulos, V. E. Silva Souza, and J. Mylopoulos, "Model Predictive Control for Software Systems with CobRA," in *SEAMS*, 2016, pp. 35–46.

[41] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "A hybrid reinforcement learning approach to autonomic resource allocation," in *ICAC*, 2006, pp. 65–73.

[42] A. Elkhodary, N. Esfahani, and S. Malek, "FUSION: a framework for engineering self-tuning self-adaptive software systems," in *FSE*, 2010, pp. 7–16.

[43] Z. Coker, D. Garlan, and C. Le Goues, "Sass: Self-adaptation using stochastic search," in *SEAMS*, 2015, pp. 168–174.