Computing (2024) 106:1251–1272
https://doi.org/10.1007/s00607-022-01052-x

SPECIAL ISSUE ARTICLE

Realizing self-adaptive systems via online reinforcement learning and feature-model-guided exploration

Andreas Metzger[1] · Clément Quinton[2] · Zoltán Ádám Mann[1] · Luciano Baresi[3] · Klaus Pohl[1]

Received: 9 March 2021 / Accepted: 20 December 2021 / Published online: 1 March 2022

Abstract

A self-adaptive system can automatically maintain its quality requirements in the presence of dynamic environment changes. Developing a self-adaptive system may be difficult due to design time uncertainty; e.g., anticipating all potential environment changes at design time is in most cases infeasible. To realize self-adaptive systems in the presence of design time uncertainty, online machine learning, i.e., machine learning at runtime, is increasingly used. In particular, online reinforcement learning is proposed, which learns suitable adaptation actions through interactions with the environment at runtime. To learn about its environment, online reinforcement learning has to select actions that were not selected before, which is known as exploration. How exploration happens impacts the performance of the learning process. We focus on two problems related to how adaptation actions are explored. First, existing solutions randomly explore adaptation actions and thus may exhibit slow learning if there are many possible adaptation actions. Second, they are unaware of system evolution, and thus may explore new adaptation actions introduced during evolution rather late. We propose novel exploration strategies that use feature models (from software product line engineering) to guide exploration in the presence of many adaptation actions and system evolution. Experimental results for two realistic self-adaptive systems indicate an average speed-up of the learning process of 33.7% in the presence of many adaptation actions, and of 50.6% in the presence of evolution.

Research leading to these results received funding from the EU Horizon 2020 programme under Grant Agreements Nos. 780351 (ENACT) and 871525 (FogProtect), and ANR-19-CE25-0003-01 KOALA project.

✉ Zoltán Ádám Mann
zoltan.mann@gmail.com

1 Paluno, Universität Duisburg-Essen, Essen, Germany

2 Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

3 Politecnico di Milano, Milan, Italy

Keywords Adaptive system · Reinforcement learning · Feature model · Evolution

Mathematics Subject Classification 68T05 · 68U01 · 68N99

1 Introduction

A *self-adaptive* system can modify its own structure and behavior at runtime based on its perception of the environment, of itself and of its requirements [9,24,34]. An example is a self-adaptive web service, which faced with a sudden increase in workload, may reconfigure itself by deactivating optional system features. An online store, for instance, may deactivate its resource-intensive recommender engine in the presence of a high workload. By adapting itself at runtime, the web service is able to maintain its quality requirements (here: performance) under changing workloads.

To develop a self-adaptive system, software engineers have to develop *self-adaptation logic* that encodes when and how the system should adapt itself. However, in doing so, software engineers face the challenge of *design time uncertainty* [6, 45]. Among other concerns, developing the adaptation logic requires anticipating the potential environment states the system may encounter at runtime to define *when* the system should adapt itself. Yet, anticipating all potential environment states is in most cases infeasible due to incomplete information at design time. As an example, take a service-oriented system which dynamically binds concrete services at runtime. The concrete services that will be bound at runtime and thus their quality are typically not known at design time. As a further concern, the precise effect of an adaptation action may not be known and thus accurately determining *how* the system should adapt itself is difficult. As an example, while software engineers may know in principle that activating more features will have a negative impact on performance, exactly determining the performance impact is more challenging [36].

Online reinforcement learning is an emerging approach to realize self-adaptive systems in the presence of design time uncertainty. Online reinforcement learning means that machine learning is employed at runtime (see existing solutions discussed in Sect. 7). The system can learn from actual operational data and thereby can leverage information only available at runtime. In general, reinforcement learning aims to learn suitable actions via an agent's interactions with its environment [37]. The agent receives a reward for executing an action. The reward expresses how suitable that action was. The goal of reinforcement learning is to optimize cumulative rewards.

1.1 Problem statement

Reinforcement learning faces the exploration–exploitation dilemma [37]. To optimize cumulative rewards, actions should be selected that have shown to be suitable, which is known as *exploitation*. However, to discover such actions in the first place, actions that were not selected before should be selected, which is known as *exploration*. How exploration happens has an impact on the performance of the learning process [4,13,

37]. We focus on two problems related to how a system's set of possible adaptation actions, i.e., its *adaptation space*, is explored.

Random exploration: Existing online reinforcement learning solutions for self-adaptive systems propose randomly selecting adaptation actions for exploration (see Sect. 7).

The effectiveness of exploration therefore directly depends on the size of the adaptation space, because each adaptation action has an equal chance of being selected. Some reinforcement learning algorithms can cope with a large space of actions, but require that the space of actions is continuous in order to generalize over unseen actions [29]. Self-adaptive systems may have large, *discrete* adaptation spaces. Examples include service-oriented systems, which may adapt by changing their service compositions [28], or reconfigurable software systems, which may adapt by changing their active set of features at runtime [23]. A simple example is a service composition consisting of eight abstract services that allows dynamically binding two concrete services each. Assuming no temporal or logical constraints on adaptation, this gives $2^8 = 256$ possible adaptations. In the presence of such large, discrete adaptation spaces, random exploration may lead to slow learning at runtime [4,13,37].

Evolution-unaware exploration: Existing online reinforcement learning solutions are unaware of system evolution [20]. They do not consider that a self-adaptive system, like any software system, typically undergoes evolution [15]. In contrast to self-adaptation, which refers to the automatic modification of the system by itself, evolution refers to the manual modification of the system [24]. Due to evolution, the adaptation space may change, e.g., existing adaptation actions may be removed or new adaptation actions may be added. Some reinforcement learning algorithms can cope with environments that change over time (non-stationary environments) [29,37]. However, they cannot cope with changes of the adaptation space. Existing solutions thus explore new adaptation actions only with low probability (as all adaptation actions have an equal chance of being selected), and thus may take quite long until new adaptation actions have been explored.

Thus, this paper addresses two problems of exploration in online reinforcement learning for self-adaptation: (1) coping with large discrete adaptation spaces and (2) coping with changes of the adaptation space due to evolution.

1.2 Contributions

We introduce exploration strategies for online reinforcement learning that address the above two problems. Our exploration strategies use *feature models* [25] to give structure to the system's adaptation space and thereby leverage additional information to guide exploration. A feature model is a tree or a directed acyclic graph of features, organized hierarchically. An adaptation action is represented by a valid feature combination specifying the target run-time configuration of the system.

Our strategies traverse the system's feature model to select the next adaptation action to be explored. By leveraging the structure of the feature model, our strategies guide the exploration process. In addition, our strategies detect added and removed

adaptation actions by analyzing the differences between the feature models of the system before and after an evolution step. Adaptation actions removed as a result of evolution are no longer explored, while added adaptation actions are explored first.

This article has been substantially extended from our earlier conference publication [26] and provides the following main new contributions:

Broader scope: We extended the scope to cover self-adaptive software systems, thereby generalizing from self-adaptive services focused in [26]. This is reflected by providing a conceptual framework for integrating reinforcement learning into the MAPE-K reference model of self-adaptive systems, adding an additional subject system from a different domain, as well as by expanding the discussion of related work.

Additional reinforcement learning algorithm: In addition to integrating our strategies into the Q-Learning algorithm, we integrate them into the SARSA algorithm. These two algorithms differ with respect to how the knowledge is updated during the learning process. Q-Learning updates the knowledge on the basis of the best possible next action. SARSA updates the knowledge on the basis of the action that the already learned policy takes [37]. As a result, Q-Learning tends to perform better in the long run. However, SARSA is better in avoiding expensive adaptations. If, for a given system, executing "wrong" adaptations is expensive, then SARSA is more appropriate, otherwise Q-Learning is preferable. Our strategies work for both algorithms.

Additional subject system: In addition to the adaptive cloud service in [26], we validate our approach with a reconfigurable data base system. The two systems differ in terms of their adaptation space, the structure of their feature model, and their quality characteristics (response time instead of energy and virtual machine migrations), thereby contributing to the external validity of our experiments.

In what follows, Sect. 2 explains fundamentals of feature models and self-adaptation, explains the integration of reinforcement learning into the MAPE-K reference model, as well as introduces a running example. Section 3 describes our exploration strategies and how they are integrated with the Q-Learning and SARSA algorithms. Section 4 presents the design of our experiments, and Sect. 5 presents our experimental results. Section 6 provides a discussion of current limitations and assumptions. Section 7 analyzes related work. Section 8 provides a conclusion and outlook on future work.

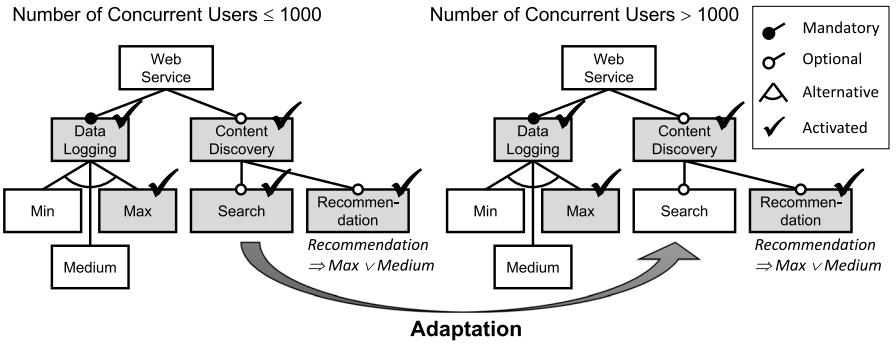

Fig. 1 Feature model and adaptation of example web service

2 Fundamentals

2.1 Feature models and self-adaptation

A *feature model* is a tree of features organized hierarchically [25] and describes the possible and allowed feature combinations. A feature f can be decomposed into mandatory, optional or alternative sub-features. If feature f is activated, its mandatory sub-features have to be activated, its optional sub-feature may or may not be activated, and at least one of its alternative sub-features has to be activated. Additional cross-tree constraints express inter-feature dependencies. A feature model can be used to define a self-adaptive system's adaptation space, where each adaptation action is expressed in terms of a possible runtime configuration, i.e., feature combination [12,16].

Figure 1 shows the feature model of a self-adaptive web service as a running example. The DataLogging feature is mandatory (which means it is always active), while the ContentDiscovery feature is optional. The DataLogging feature has three alternative sub-features, i.e., at least one data logging sub-feature must be active: Min, Medium or Max. The ContentDiscovery feature has two optional sub-features Search and Recommendation. The cros-tree constraint Recommendation $\Rightarrow$ Max $\vee$ Medium specifies that a sufficient level of data logging is required to collect enough information about the web service's users and transactions to make good recommendations.

Let us consider that the above web service should adapt to a changing number of concurrent users to keep its response time below 500 ms. A software engineer may express an adaptation rule for the web service such that it turns off some of its features in the presence of more users, thereby reducing the resource needs of the service. The right-hand side of Fig. 1 shows a concrete example for such an adaptation. If the service faces an environment state of more than 1000 concurrent users, the service self-adapts by deactivating the Search feature.

2.2 Reinforcement learning and self-adaptation

As illustrated in Fig. 2a, reinforcement learning aims to learn an optimal action selection policy via an agent's interactions with its environment [37].

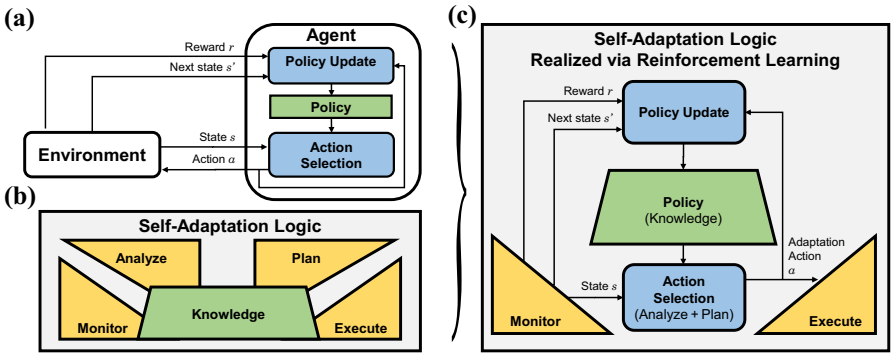

Fig. 2 Integration of reinforcement learning into the MAPE-K reference model: **a** basic reinforcement learning model, **b** MAPE-K model, **c** integrated model

At a given time step t, the agent selects an action a (from its adaptation space) to be executed in environment state s. As a result, the environment transitions to s' at time step $t + 1$ and the agent receives a reward r for executing the action. The reward r together with the information about the next state s' are used to update the action selection policy of the agent. The goal of reinforcement learning is to optimize cumulative rewards. As mentioned in Sect. 1, a trade-off between *exploitation* (using current knowledge) and *exploration* (gathering new knowledge) must be made. That is, to optimize rewards, actions should be selected that have shown to be useful (*exploitation*), but to discover such actions in the first place, actions that were not selected before must also be selected (*exploration*).

A self-adaptive system can conceptually be structured into two main elements [19,34]: the *system logic* (aka. the managed element) and the *self-adaptation logic* (aka. the autonomic manager). To understand how reinforcement learning can be leveraged for realizing the self-adaptation logic, we use the well-established MAPE-K reference model for self-adaptive systems [9,44]. As depicted in Fig. 2b, MAPE-K structures the self-adaptation logic into four main conceptual activities that rely on a common *knowledge* base [17]. These activities *monitor* the system and its environment, *analyze* monitored data to determine adaptation needs, *plan* adaptation actions, and *execute* these adaptation actions at runtime.

Figure 2c depicts how the elements of reinforcement learning are integrated into the MAPE-K loop.

For a self-adaptive system, "agent" refers to the self-adaptation logic of the system and "action" refers to an adaptation action [30]. In the integrated model, *action selection* of reinforcement learning takes the place of the *analyze* and *plan* activities of MAPE-K. The learned *policy* takes the place of the self-adaptive system's *knowledge* base. At runtime, the policy is used by the self-adaptation logic to select an adaptation action a based on the current state s determined by *monitoring*. The action selected using the policy may be either to leave the system in the current state (i.e., no need for adaptation), or a specific adaptation, which is then *executed*.

3 Feature-model-guided exploration (FM-guided exploration)

As motivated in Sect. 1, our exploration strategies use feature models (FM) to guide the exploration process. We first explain how these FM-guided exploration strategies can be integrated into existing reinforcement learning algorithms. Thereby, we also provide a realization of the integrated conceptual model from Sect. 2. We then introduce the realization of the actual FM-guided exploration strategies.

3.1 Integration into reinforcement learning

We use two well-known reinforcement learning algorithms for integrating our FM-guided exploration strategies: Q-Learning and SARSA. We chose Q-Learning, because it is the most widely used algorithm in the related work (see Sect. 7). We chose SARSA, as it differs from Q-Learning with respect to how the knowledge is updated during learning. Q-Learning (an *off*-policy algorithm) updates the knowledge based on selecting the next action which has the highest expected reward [37]. SARSA (an *on*-policy algorithm) updates the knowledge based on selecting the next action by following the already learned action selection policy.

Algorithm 1 shows the extended Q-Learning algorithm. A value function $Q(s, a)$ represents the learned knowledge, which gives the expected cumulative reward when performing an action a in a state s [37]. There are two hyper-parameters: the learning rate α, which defines to what extent newly acquired knowledge overwrites old knowledge, and the discount factor γ, which defines the relevance of future rewards. After the initialization (lines 2–3), the algorithm repeatedly selects the next action (line 5), performs the action and observes its results (line 6), and updates its learned knowledge and other variables (lines 7–8). Algorithm 2 shows the extended SARSA algorithm, which follows a similar logic. However, while Q-Learning updates the knowledge by selecting the action with the highest Q value (Algorithm 1, line 7), SARSA selects the action according to the current policy (Algorithm 2, line 8).

Our strategies are integrated into reinforcement learning in the getNextAction function, which selects the next adaptation action while trading off exploration and exploitation. We use the ϵ-greedy strategy as a baseline, as a standard action selection strategy in reinforcement learning, widely used in the related work (see Sect. 7). With probability $1 - \epsilon$, ϵ-greedy exploits existing knowledge, while with probability ϵ, it selects a random action. In contrast to random exploration, we use our FM-guided exploration strategies by calling the getNextConfiguration function (Algorithm 1, line 17). To prevent FM-guided exploration from prematurely converging to a local minimum, we follow the literature and use a little randomness [31], i.e., perform random exploration with probability $\delta \cdot \epsilon$ (lines 15, 16). Here, $0 \leq \delta \leq 1$ is the probability for choosing a random action, given that we have chosen to perform exploration.

To facilitate convergence of the learning process, we use the ϵ-decay approach. This is a typical approach in reinforcement learning, which starts at $\epsilon = 1$ and decreases it at a predefined rate ϵ_d after each time step. We also follow this decay approach for the FM-guided strategies to incrementally decrease δ with rate δ_d.

Algorithm 1 Q-Learning with FM-guided Exploration

```
1: function FMQ- LEARNING(FeatureModel 𝓜; Double α, γ, ϵ_d, δ_d)
2:     Initialize Q(s, a) with lowest possible reward ∀s ∈ S (state space), ∀a ∈ A (adaptation space);
3:     Determine current state s; ϵ ← 1; δ ← 1;
4:     repeat
5:         Set<Feature> a = GETNEXTACTION(𝓜, s); // Action Selection
6:         Adapt service to configuration a; Observe reward r; Observe new state s′;
7:         Q(s, a) ← Q(s, a) + α[r + γ max_{a′∈A} Q(s′, a′) − Q(s, a)]; // Knowledge Update
8:         s ← s′; ϵ ← ϵ · ϵ_d; δ ← δ · δ_d;
9:     until last time step
10: end function
11:
12: function GETNEXTACTION(FeatureModel 𝓜, State s)
13:     Set<Feature> a ← argmax_a Q(s, a); // Exploit existing knowledge
14:     INITFMEXPLORATION(𝓜, a); // initialize the FM-guided strategies, see Algorithm 3
15:     if random() < ϵ then // Explore new actions
16:         if random() < δ then return GETRANDOMCONFIGURATION(𝓜);
17:         else return GETNEXTCONFIGURATION(); // see Algorithm 3
18:         end if
19:     end if
20:     return a;
21: end function
```

Algorithm 2 SARSA with FM-guided Exploration

```
1: function FMSARSA(FeatureModel 𝓜; Double α, γ, ϵ_d, δ_d)
2:     Initialize Q(s, a) with lowest possible reward ∀s ∈ S (state space), ∀a ∈ A (adaptation space);
3:     Determine current state s; ϵ ← 1; δ ← 1;
4:     Set<Feature> a = GETNEXTACTION(𝓜, s); // Action Selection
5:     repeat
6:         Adapt service to configuration a; Observe reward r; Observe new state s′;
7:         Set<Feature> a′ = GETNEXTACTION(𝓜, s′);
8:         Q(s, a) ← Q(s, a) + α[r + γ Q(s′, a′) − Q(s, a)]; // Knowledge Update
9:         s ← s′; a ← a′; ϵ ← ϵ · ϵ_d; δ ← δ · δ_d;
10:    until last time step
11: end function
```

3.2 Feature–model–structure exploration for large adaptation spaces

To capture large adaptation spaces, we propose the FM-structure exploration strategy, which takes advantage of the semantics typically encoded in the structure of feature models. Non-leaf features are typically abstract features used to better structure variability [40]. Abstract features do not directly impact the implementation, but delegate their implementation to their sub-features. Sub-features thereby offer different implementations of their abstract parent feature. As such, the sub-features of a common parent feature, i.e., *sibling* features, can be considered semantically connected.

In the example from Sect. 2, the ContentDiscovery feature has two sub-features Search and Recommendation offering different concrete ways how a user may discover online content. The idea behind FM-structure exploration is to exploit the information about these potentially semantically connected sibling features and explore them first before

Table 1 Example for FM-structure exploration (excerpt)

	Logging	Min	Medium	Max	Content Disc.	Search	Recommend.
Start	✓			✓	✓		✓
1	✓		✓		✓		✓
2	✓		✓		✓	✓	✓
3	✓			✓	✓	✓	✓
4	✓		✓		✓	✓	
5	✓	✓			✓	✓	
	...	...	...	...	...	...	...

exploring other features.[1] Table 1 shows an excerpt of a typical exploration sequence of the FM-structure exploration strategy with the step-wise exploration of sibling features highlighted in gray. Exploration starts with a randomly selected leaf feature, here: Recommendation. Then all configurations involving this leaf feature are explored before moving to its sibling feature, here: Search.

FM-structure exploration is realized by Algorithm 3, which starts by randomly selecting an arbitrary leaf feature f among all leaf features that are part of the current configuration (lines 5, 6). Then, the set of configurations $\mathscr{C}_f$ containing feature f is computed, while the sibling features of feature f are gathered into a dedicated *siblings* set (line 7). While $\mathscr{C}_f$ is non-empty, the strategy explores one randomly selected configuration from $\mathscr{C}_f$ and removes the selected configuration from $\mathscr{C}_f$ (lines 11–13). If $\mathscr{C}_f$ is empty, then a new set of configurations containing a sibling feature of f is randomly explored, provided such sibling feature exists (lines 15–17). If no configuration containing f or a sibling feature of f is found, the strategy moves on to the parent feature of f, which is repeated until a configuration is found (line 13) or the root feature is reached (line 22).

3.3 Feature–model–difference exploration strategy for system evolution

To capture changes in the system's adaptation space due to system evolution, we propose the FM-difference exploration strategy, which leverages the differences in feature models before ($\mathscr{M}$) and after ($\mathscr{M}'$) an evolution step. Following the product line literature, we consider two main types of feature model differences [39]:

Added configurations (feature model generalization). New configurations may be added to the adaptation space by (*i*) introducing new features to $\mathscr{M}'$, or (*ii*) removing or relaxing existing constraints (e.g., by changing a sub-feature from mandatory to optional, or by removing cross-tree constraints). In our running example, a new sub-feature Optimized might be added to the DataLogging feature, providing a more resource efficient logging implementation. Thereby, new configurations are added to the adaptation space, such as {DataLogging, Optimized, ContentDiscovery, Search}. As another example, the Recommendation implementation may have been improved and it now can work with the Min logging feature. This removes the cross-tree constraint shown in Fig. 1, and adds new configurations such as {DataLogging, Min, ContentDiscovery, Recommendation}.

[1] Note that this entails a random selection of the order of sub-features.

Algorithm 3 FM-Structure Exploration Strategy

```
1: Set<Feature> leaves, configuration, siblings;
2: Set<Set<Feature>> 𝒞_f; Feature f;
3:
4: function InitFMExploration(FeatureModel ℳ, Set<Feature> currentConfiguration)
5:     leaves ← getLeaves(currentConfiguration);
6:     f ← randomSelect(leaves);
7:     𝒞_f ← getConfigurationsWithFeature(f); siblings ← siblings(f);
8: end function
9:
10: function getNextConfiguration()
11:     if 𝒞_f ≠ ∅ then
12:         configuration ← randomSelect(𝒞_f); 𝒞_f ← 𝒞_f \ {configuration};
13:         return configuration;
14:     else
15:         if siblings ≠ ∅ then
16:             f ← randomSelect(siblings);
17:             siblings ← siblings \ {f}; 𝒞_f ← getConfigurationsWithFeature(f);
18:         else
19:             if parent(f) ≠ ∅ then
20:                 f ← parent(f); siblings ← siblings(f);
21:                 𝒞_f ← getConfigurationsWithFeature(f);
22:             else // Root feature reached
23:                 return ∅;
24:             end if
25:         end if
26:         return getNextConfiguration();
27:     end if
28: end function
```

Removed configurations (feature model specialization). Symmetrical to above, configurations may be removed from the adaptation space by (*i*) removing features from $\mathscr{M}$, or (*ii*) by adding or tightening constraints in $\mathscr{M}'$.

To determine these changes of feature models, we compute a set-theoretic difference between valid configurations expressed by feature model $\mathscr{M}$ and feature model $\mathscr{M}'$. Detailed descriptions of feature model differencing as well as efficient tool support can be found in [1,5]. The feature model differences provide us with adaptation actions added to the adaptation space ($\mathscr{M}' \setminus \mathscr{M}$), as well as adaptation actions removed from the adaptation space ($\mathscr{M} \setminus \mathscr{M}'$).

Our FM-difference exploration strategies first explore the configurations that were added to the adaptation space, and then explore the remaining configurations if needed. The rationale is that added configurations might offer new opportunities for finding suitable adaptation actions and thus should be explored first. Configurations that were removed are no longer executed and thus the learning knowledge can be pruned accordingly. In the reinforcement learning realization (Sect. 3.1), we remove all tuples (s, a) from Q, where a represents a removed configuration.

FM-difference exploration can be combined with FM-structure exploration, but also with ϵ-greedy. In both cases, this means that instead of exploring the whole new adaptation space, exploration is limited to the set of new configurations.

4 Experiment setup

We experimentally assess our FM-guided exploration strategies and compare them with ϵ-greedy as the strategy used in the related work (see Sect. 7). In particular, we aim to answer the following research questions:

RQ1: How does learning performance and system quality using FM-structure exploration (from Sect. 3.2) compare to using ϵ-greedy?

RQ2: How does learning performance and system quality using FM-difference exploration (from Sect. 3.3) compare to evolution-unaware exploration?

4.1 Subject systems

Our experiments build on two real-world systems and datasets. The CloudRM system is an adaptive cloud resource management service offering 63 features, 344 adaptation actions, and a feature model that is 3 levels deep. The BerkeleyDB-J system is an open source reconfigurable database written in Java with 26 features, 180 adaptation actions and 5 levels.

CloudRM System: CloudRM [21] controls the allocation of computational tasks to virtual machines (VMs) and the allocation of virtual machines to physical machines in a cloud data center.[2] CloudRM can be adapted by reconfiguring it to use different allocation algorithms, and the algorithms can be adapted by using different sets of parameters. We implemented a separate adaptation logic for CloudRM by using the extended learning algorithms as introduced in Sect. 3.1.

We define the reward function as $r = -(\rho \cdot e + (1-\rho) \cdot m)$, with energy consumption e and number of VM manipulations m (i.e., migrations and launches), each normalized to be within $[0, 1]$. We use $\rho = 0.8$, meaning we give priority to reducing energy consumption, while still maintaining a low number of VM manipulations.

Our experiments are based on a real-world workload trace with 10,000 tasks, in total spanning over a time frame of 29 days [22]. The CloudRM algorithms decide on the placement of new tasks whenever they are entered into the system (as driven by the workload trace). For RQ2, the same workload was replayed after each evolution step to ensure consistency among the results.

To emulate system evolution, we use a 3-step evolution scenario.

Starting from a system that offers 26 adaptation actions, these three evolution steps respectively add 30, 72 and 216 adaptation actions.

BerkeleyDB-J: The BerkeleyDB-J dataset was collected by Siegmund et al [36] and was used for experimentation with reconfigurable systems to predict their response times.[3] We chose this system because the configurations are expressed as a feature model and the dataset includes performance measurements for all system configurations, which were measured using standard benchmarks.[4] Adaptation actions are the

[2] https://sourceforge.net/p/vm-alloc/task_vm_pm.

[3] https://www.se.cs.uni-saarland.de/projects/splconqueror/icse2012.php.

[4] Other datasets from [36] had feature models with only 1 level, had many configurations associated with the same response time, or did not include performance measurements for all configurations.

possible runtime reconfigurations of the system. We define the reward function as $r = -t$, with t being the response time normalized to be within [0, 1].[5]

Given the smaller size of BerkeleyDB-J's adaptation space, we use a 2-step evolution scenario to emulate system evolution. We first randomly change two of the five optional features into mandatory ones, thereby reducing the size of the adaptation space. We start from this reduced adaptation space and, randomly change the mandatory features back into optional ones. Starting from a system that offers 39 adaptation actions, these two evolution steps respectively add 20 and 121 adaptation actions.

4.2 Measuring learning performance

We characterize the performance of the learning process by using the following metrics from [38]: *Asymptotic performance* measuring the reward achieved at end of the learning process. *Time to threshold* measuring the number of time steps it takes the learning process to reach a predefined reward threshold (in our case 90% of the difference between maximum and minimum performance). *Total performance* measuring the overall learning performance by computing the area between the reward curve and the asymptotic reward. In addition, we measure how the different strategies impact on the quality characteristics of the subject systems.

Given the stochastic nature of the learning strategies (both ϵ-greedy and to a lesser degree our strategies involve random decisions), we repeated the measurements 500 times and averaged the results.

4.3 Prototypical realization

The learning algorithms, as well as the ϵ-greedy and FM-based exploration strategies were implemented in Java. Feature model management and analysis were performed using the FeatureIDE framework,[6] which we used to efficiently compute possible feature combinations from a feature model.

4.4 Hyper-parameter optimization

To determine suitable hyper-parameter values (see Sect. 3.1), we performed hyper-parameter tuning via exhaustive grid search for each of the subject systems and each of the reinforcement learning algorithms. We measured the learning performance for our baseline ϵ-greedy strategy for 11,000 combinations of learning rate α, discount factor γ, and ϵ-decay rate. For each of the subject systems and reinforcement learning algorithms we chose the hyper-parameter combination that led to the highest asymptotic performance. We used these hyper-parameters also for our FM-guided strategies.

[5] For CloudRM, the reward function was the opposite of the weighted sum of the metrics to be minimized, where the sum of the weights is 1. Regarding BerkeleyDB-J, the same logic is applied, but since there is only one metric to be minimized, the formula becomes simpler.

[6] https://featureide.github.io/.

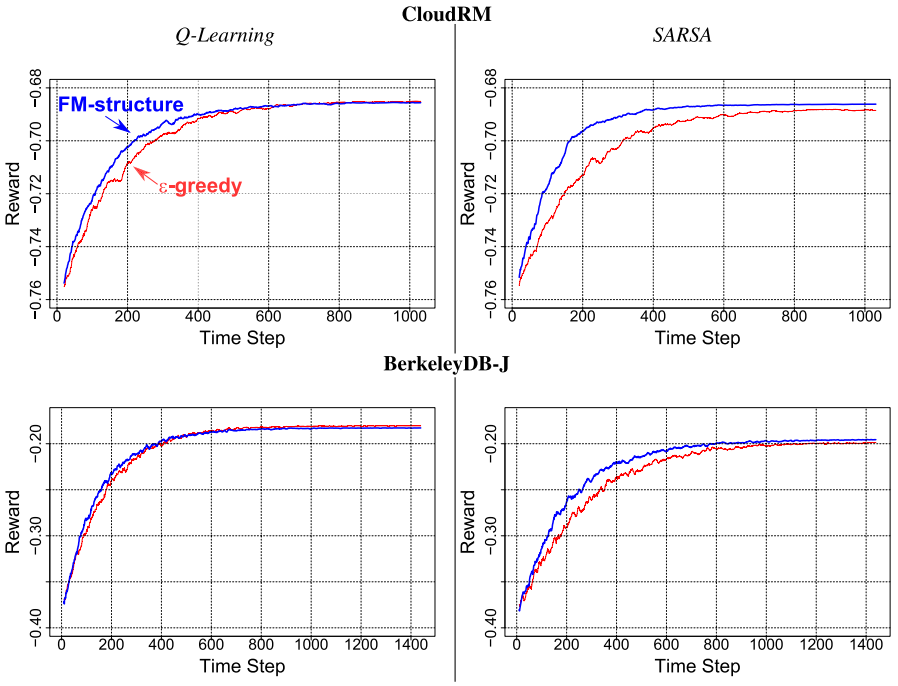

Fig. 3 Learning performance for large adaptation spaces (RQ1)

5 Results

To facilitate reproducibility and replicability, our code, the used data and our experimental results are available online.[7]

5.1 Results for RQ1 (FM-structure exploration)

Figure 3 visualizes the learning process by showing how rewards develop over time, while Table 2 quantifies the learning performance using the metrics introduced above.

Across the two systems and learning algorithms, FM-structure exploration performs better than ϵ-greedy wrt. total performance (33.7% on average) and time to threshold (25.4%), while performing comparably wrt. asymptotic performance (0.33%). A higher improvement is visible for CloudRM than for BerkeleyDB-J, which we attribute to the much larger adaptation space of CloudRM, whereby the effects of systematically exploring the adaptation space become more pronounced.

For CloudRM, FM-structure exploration consistently leads to less VM manipulations and lower energy consumption. While savings in energy are rather small (0.1% resp. 0.23%), FM-structure exploration reduces the number of virtual machine manipulations by 7.8% resp. 9.15%. This is due to the placement algorithms of CloudRM having a small difference wrt. energy optimization, but a much larger difference wrt. the

[7] https://gitlab.com/cquinton/fmlearning.

Table 2 Comparison of exploration strategies for large adaptation spaces (RQ1)

	Asymptotic performance	Time to Threshold	Total performance	Effect on Quality	
Q-Learning		**CloudRM**		Energy	VM Manip.
ϵ-greedy:	-0.6851	286	-8.8023	7084	2281
FM-structure:	-0.6854	219	-5.4431	7077	2103
Improvement	*-0.04 %*	*23.43 %*	*38.16 %*	*0.10 %*	*7.80 %*
SARSA					
ϵ-greedy:	-0.6885	390	-11.631	10602	3398
FM-structure:	-0.6862	200	-4.9673	10578	3087
Improvement	*0.33 %*	*48.72 %*	*57.29 %*	*0.23 %*	*9.15 %*
Q-Learning		**BerkeleyDB-J**		Avg. Response Time	
ϵ-greedy:	-0.1834	383	-31.2457	3606	
FM-structure:	-0.1847	357	-28.0466	3550	
Improvement	*-0.71 %*	*6.79 %*	*10.24 %*	*1.55 %*	
SARSA					
ϵ-greedy:	-0.1993	592	-46.8978	3824	
FM-structure:	-0.1958	457	-33.2211	3666	
Improvement	*1.76 %*	*22.80 %*	*29.16 %*	*4.13 %*	
Avg. Improv. Q-Learning	*-0.38 %*	*15.1 %*	*24.2 %*		
Avg. Improv. SARSA	*1.05 %*	*35.8 %*	*43.2 %*		
Total Avg. Improvement	***0.33 %***	***25.4 %***	***33.7 %***		

number of virtual machine manipulations. For BerkeleyDB-J, we observe an improvement in response times of 1.55% resp. 4.13%. This smaller improvement is consistent with the smaller improvement in learning performance.

Analyzing the improvement of FM-structure exploration for the different learning algorithms, we observe an improvement of 24.2% (total performance) resp. 15.1% (time to threshold) for Q-Learning, and a much higher improvement of 43.2% resp. 35.8% for SARSA. Note, however, that the overall learning performance of SARSA is much lower than that of Q-Learning. SARSA performs worse wrt. total performance (−23% on average), time to threshold (−27.6% on average), and asymptotic performance (−3.82% on average). In addition, SARSA requires around 19.4% more episodes than Q-Learning to reach the same asymptotic performance. The reason is that SARSA is more conservative during exploration [37]. If there is an adaptation action that leads to a large negative reward which is close to an adaptation action that leads to the optimal reward, Q-Learning exhibits the risk of choosing the adaptation action with the large negative reward. In contrast, SARSA will avoid that adaptation action, but will more slowly learn the optimal adaptation actions. So, in practice one may choose between Q-Learning and SARSA depending on how expensive it is to execute "wrong" adaptations.

5.2 Results for RQ2 (FM-difference exploration)

We compare FM-difference exploration combined with ϵ-greedy and FM-structure exploration with their respective evolution-unaware counterparts (i.e., the strategies

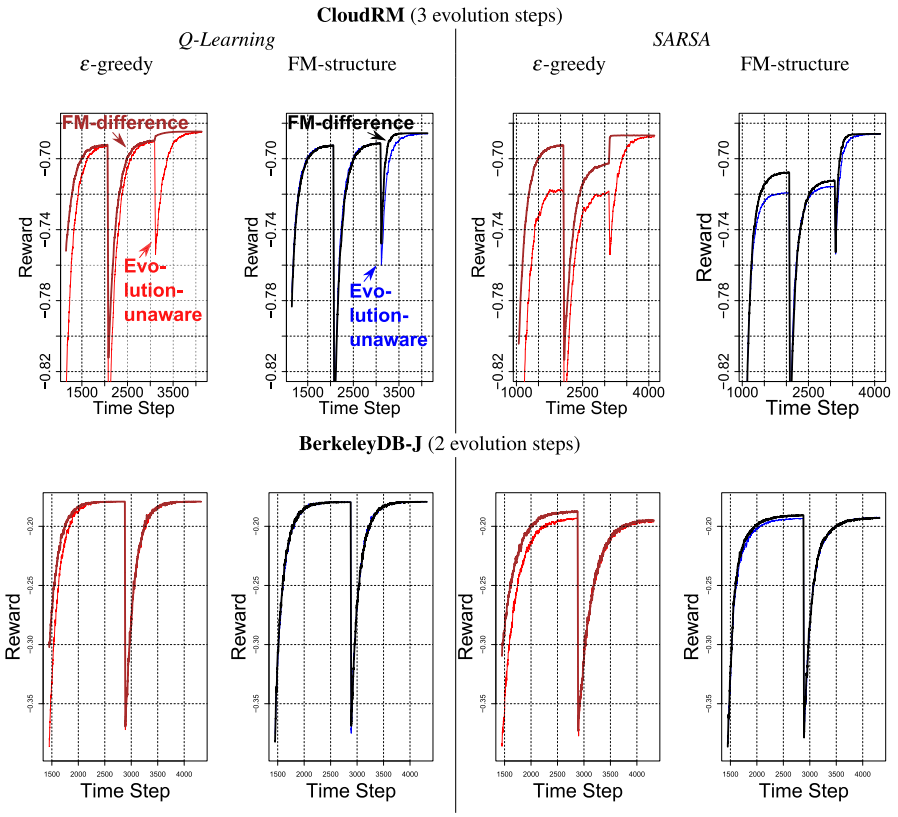

Fig. 4 Learning performance across system evolution (RQ2)

used for RQ1). It should be noted that even though we provide the evolution-unaware strategies with the information about the changed adaptation space (so they can fully explore it), we have not modified them such as to differentiate between old and new adaptation actions.

Like for RQ1, Fig. 4 visualizes the learning process, while Table 3 quantifies learning performance. We computed the metrics separately for each of the evolution steps and report their averages. After each evolution step, learning proceeds for a given number of time steps, before moving to the next evolution step.

The FM-difference exploration strategies consistently perform better than their evolution-unaware counterparts wrt. total performance (50.6% on average) and time to threshold (47%), and perform comparably wrt. asymptotic performance (1.7%). Like for RQ1, the improvements are more pronounced for CloudRM, which exhibits a larger action space than BerkeleyDB-J.

For CloudRM, FM-difference exploration reduces the number of virtual machine manipulations by 19.8% resp. 30.9%, while keeping energy consumption around the same as the non-evolution-aware strategies. For BerkeleyDB-J, FM-difference explo-

Table 3 Comparison of exploration strategies across evolution steps (RQ2)

	Asymptotic performance	Time to Threshold	Total performance	Effect on Quality	
Q-Learning		**CloudRM**		Energy	VM Manip.
ϵ-greedy:					
FM-difference	-2.0670	571	-39.1095	32393	10074
Evolution-unaware	-2.0688	1147	-84.9052	32147	13745
FM-structure:					
FM-difference	-2.0697	756	-57.5157	32351	11439
Evolution-unaware	-2.0699	866	-59.0660	32273	11798
Avg. Improvement	*0.05 %*	*57.7 %*	*59.9 %*	*-0.50 %*	*19.8 %*
SARSA					
ϵ-greedy:					
FM-difference	-2.1489	607	-39.3645	32566	10374
Evolution-unaware	-2.2530	2018	-117.6577	32618	15756
FM-structure:					
FM-difference	-2.1723	955	-74.8560	32660	12641
Evolution-unaware	-2.1834	723	-86.3582	32695	13911
Avg. Improvement	*2.68 %*	*104.1 %*	*107.1 %*	*0.13 %*	*30.9 %*

Q-Learning		**BerkeleyDB-J**		Avg. Response Time
ϵ-greedy:				
FM-difference	-0.3583	661	-52.7786	3270
Evolution-unaware	-0.3582	774	-66.2526	3348
FM-structure:				
FM-difference	-0.3589	675	-58.2346	3301
Evolution-unaware	-0.3588	693	-58.7939	3305
Avg. Improvement	*-0.02 %*	*9.88 %*	*13.2 %*	*1.24 %*
SARSA				
ϵ-greedy:				
FM-difference	-0.5111	999	-74.5953	3588
Evolution-unaware	-0.5465	1195	-101.2446	3741
FM-structure:				
FM-difference	-0.4685	726	-61.6969	3514
Evolution-unaware	-0.4732	818	-66.8781	3544
Avg. Improvement	*3.97 %*	*16.2 %*	*22.1 %*	*2.56 %*
Avg. Improv. ϵ-greedy	*2.96 %*	*92.5 %*	*94.3 %*	
Avg. Improv. FM-structure	*0.38 %*	*1.4 %*	*6.85 %*	
Total Avg. Improvement	***1.67 %***	***47 %***	***50.6 %***	

ration leads a reduction in response time by 1.24% resp. 2.56%. Like for RQ1, this smaller reduction is consistent with the smaller learning performance.

The improvement of FM-difference exploration is more pronounced for ϵ-greedy than for FM-structure exploration; e.g., showing a 94.4% improvement in total performance for ϵ-greedy compared with an improvement of only 6.85% for FM-structure exploration. This suggests that, during evolution, considering the changes of the adaptation space has a much larger effect than considering the structure of the adaptation space. In addition, we note that due to the way we emulate evolution in our experiments, the number of adaptations introduced after an evolution step is much smaller

(66 on average) than the size of the whole adaptation space of the subject systems (262 on average), thus diminishing the effect of FM-structure exploration.

Analyzing the improvement of FM-difference exploration for the different learning algorithms, we can observe the same effect as for RQ1. While FM-difference exploration shows a much higher improvement for SARSA, the overall learning performance for SARSA is much lower than for Q-Learning.

5.3 Validity risks

We used two realistic subject systems and employed real-world workload traces and benchmarks to measure learning performance and the impact of the different exploration strategies on the systems' quality characteristics. The results reinforce our earlier findings from [26] and also indicate that the size of the adaptation space may have an impact on how much improvement may be gained from FM-structure exploration. As part of our future work, we plan experiments with additional subject systems to confirm this impact for larger action spaces.

We chose ϵ-greedy as a baseline, because it was the exploration strategy used in existing online reinforcement learning approaches for self-adaptive systems (see Sect. 7). Alternative exploration strategies were proposed in the broader field of machine learning. Examples include Boltzmann exploration, where actions with a higher expected reward (e.g., Q value) have a higher chance of being explored, or UCB action selection, where actions are favored that have been less frequently explored [37]. A comparison among those alternatives is beyond the scope of this article, because a fair comparison would require the careful variation and analysis of a range of many additional hyper-parameters. We plan addressing this as part of future work.

6 Limitations and assumptions

Below, we discuss current limitations and assumptions of our approach.

6.1 Completeness of feature models

We assume that feature models are complete with respect to the coverage of the adaptation space and that during an evolution step they are always consistent and up to date. A further possible change during service evolution can be the modification of a feature's implementation, which is currently not visible in the feature models. Encoding such kind of modification thus could further improve our FM-guided exploration strategies.

6.2 Structure of feature models

One aspect that impacts FM-structure exploration is how the feature model is structured. As an example, if a feature model has only few levels (and thus little structure), FM-structure exploration behaves similar to random exploration, because such a "flat"

feature model does not provide enough structural information. On the other hand, providing reinforcement learning with too much structural information might hinder the learning process. As case in point, we realized during our experiments that the alternative FM-structure exploration strategy from our earlier work [26] indeed had such negative effect for the BerkeleyDB-J system. This alternative strategy used the concept of "feature degree"[8] to increase the amount of structural information used during learning.

6.3 Types of features

Our approach currently only supports discrete features in the feature models, and thus only discrete adaptation actions. Capturing feature cardinalities or allowing numeric feature values is currently not possible, and thus continuous adaptation actions cannot be captured.

6.4 Adaptation constraints

When realizing the exploration strategies (both ϵ-greedy and FM-guided), we assumed we can always switch from a configuration to any other possible configuration. We were not concerned with the technicalities of how to reconfigure the running system (which, for example, is addressed in [8]). We also did not consider constraints concerning the order of adaptations. In practice, only certain paths may be permissible to reach a configuration from the current one. To consider such paths, our strategies may be enhanced by building on work such as [32].

7 Related work

We first review papers that apply online reinforcement learning to self-adaptive systems but do not consider large discrete adaptation spaces or system evolution, and then review papers that do.

7.1 Applying online reinforcement learning to self-adaptive systems

Barrett et al use Q-Learning with ϵ-greedy for autonomic cloud resource allocation [3]. They use parallel learning to speed up the learning process. Caporuscio et al use two-layer hierarchical reinforcement learning for multi-agent service assembly [7]. They observe that by sharing monitoring information, learning happens faster than when learning in isolation. Arabnejad et al apply fuzzy reinforcement learning with ϵ-greedy to learn fuzzy adaptation rules [2]. Moustafa and Zhang use multi-agent Q-Learning with ϵ-greedy for adaptive service compositions [28]. To speed up learning, they use collaborative learning, where multiple systems simultaneously explore the set of concrete services to be composed. Zhao et al use reinforcement learning (with

[8] The feature degree for a given feature f is the number of configurations that contain f.

ϵ-greedy) combined with case-based reasoning to generate and update adaptation rules for web applications [46]. Their approach may take as long to converge as learning from scratch, but may offer higher initial performance. Shaw et al apply reinforcement learning for the consolidation of virtual machines in cloud data centers [35].

Recently, deep reinforcement learning has gained popularity. In deep reinforcement learning a deep neural network is used to store the learned knowledge (for example, the Q function in Deep Q-Learning). Wang et al use Q-learning (using ϵ-greedy) together with function approximation. They use neural networks to generalize over unseen environment states and thereby facilitate learning in the presence of many environment states [42]. Yet, they do not address large action spaces. Moustafa and Ito use deep Q-Networks enhanced with double Q-Learning and prioritized experience replay for adaptively selecting web services for a service workflow [27]. Wang et al also address the service composition problem, and apply deep Q-Learning with Recurrent Neural Network (RNN), which can also handle partially observable states [43]. Restuccia and Melodia propose an efficient implementation of reinforcement learning using deep Q-Networks for adaptive radio control in wireless embedded systems [33]. In our earlier work, we used policy-based reinforcement learning for self-adaptive information systems [30], where the policy is represented as a neural network. Thereby, we addressed continuous environment states and adaptation actions. Using deep neural networks, these approaches can better generalize over environment states and actions. Thereby, deep reinforcement learning in general may perform better in large adaptation spaces. However, to be able to generalize, the adaptation space must be continuous [10]. A continuous space of actions is represented by continuous variables, such as real-valued variables. Setting a specific angle for a robot arm or changing the set-point of a thermostat are examples for a continuous space of actions [18]. However, as motivated in Sect. 1, many kinds of self-adaptive systems have a discrete, i.e., non-continuous space of adaptation actions.

In conclusion, none of the approaches reviewed above addresses the influence of large discrete adaptation spaces nor that of system evolution on learning performance. Thus, these approaches may suffer from poor performance over an extended period of time while the system is performing random exploration of the large adaptation space, as it may take a long time to find suitable adaptations. In addition, such approaches may only recognize adaptation possibilities added by evolution late, thereby negatively impacting on the system's long-term overall performance.

7.2 Considering large adaptation spaces and evolution

Bu et al explicitly consider large adaptation spaces [4]. They employ Q-Learning (using ϵ-greedy) for self-configuring cloud systems. They reduce the size of the adaptation space by splitting it into coarse-grained sub-sets for each of which they find a representative adaptation action using the simplex method. Their experiments indicate that their approach indeed can speed up learning. Yet, they do not consider service evolution.

Dutreilh et al explicitly consider service evolution [11]. They use Q-Learning for autonomic cloud resource management. To speed up learning, they provide good ini-

tial estimates for the Q function, and use statistical estimates about the environment behavior. They indicate that system evolution may imply a change of system performance and sketch an idea on how to detect such drifts in system performance. Yet, they do not consider that evolution may also introduce or remove adaptation actions.

A different line of work uses supervised machine learning to reduce the size of the adaptation space. As an example, Van Der Donckt et al use deep learning to determine a representative and much smaller subset of the adaptation space [41]. However, supervised learning requires labeled training data representative of the system's environment, which may be challenging to obtain due to design time uncertainty.

Our earlier work made a first attempt to address both large adaptation spaces and evolution in online reinforcement learning for self-adaptive systems by means of FM-guided exploration [26]. The present paper extends our earlier work in multiple respects, including a conceptual framework for integrating reinforcement learning into the MAPE-K reference model of self-adaptive systems, covering a broader range of subject systems, and integrating our strategies into two different reinforcement learning algorithms.

8 Conclusion and outlook

We introduced feature-model-guided exploration strategies for online reinforcement learning that address potentially large adaptation spaces and the change of the adaptation space due to system evolution. Experimental results for two adaptive systems indicate a speed up of learning and an improvement of quality characteristics in turn.

As future work, we plan using extended feature models, which offer a more expressive notation allowing to capture feature cardinality and even numeric feature values [25]. Thereby, we can express adaptation spaces which combine discrete and continuous adaptation actions. This will require more advanced feature analysis methods [14] to be used as part of our FM-based exploration strategies. In addition, we aim to extend our strategies to also consider changes in existing features (and not only additions and removal of features) during system evolution. This, among others, will require extending the feature modeling language used. Finally, we plan using deep reinforcement learning, which represents the policy as a neural network, and thereby can generalize over environment states and adaptation actions [30].

Acknowledgements We cordially thank Antonio Ruiz-Cortés for inspiring discussions at ICSOC 2020, as well as Alexander Palm for comments on earlier drafts.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Acher M, Heymans P, Collet P, Quinton C, Lahire P, Merle P (2012) Feature model differences. In: Proceedings of the 24th international conference on advanced information systems engineering, CAiSE'12, pp 629–645
2. Arabnejad H, Pahl C, Jamshidi P, Estrada G (2017) A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling. In: 17th intl symposium on cluster, cloud and grid computing, CCGRID 2017, pp 64–73
3. Barrett E, Howley E, Duggan J (2013) Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. Concurr Comput Pract Exp 25(12):1656–1674
4. Bu X, Rao J, Xu C (2013) Coordinated self-configuration of virtual machines and appliances using a model-free learning approach. IEEE Trans Parallel Distrib Syst 24(4):681–690
5. Bürdek J, Kehrer T, Lochau M, Reuling D, Kelter U, Schürr A (2016) Reasoning about product-line evolution using complex feature model differences. Autom Softw Eng 23(4):687–733
6. Calinescu R, Mirandola R, Perez-Palacin D, Weyns D (2020) Understanding uncertainty in self-adaptive systems. In: IEEE international conference on autonomic computing and self-organizing systems, ACSOS 2020, Washington, DC, USA, August 17–21, 2020, pp 242–251. IEEE
7. Caporuscio M, D'Angelo M, Grassi V, Mirandola R (2016) Reinforcement learning techniques for decentralized self-adaptive service assembly. In: 5th Eur. conference on service-oriented and cloud computing, ESOCC'16, vol 9846, pp 53–68
8. Chen B, Peng X, Yu Y, Nuseibeh B, Zhao W (2014) Self-adaptation through incremental generative model transformations at runtime. In: 36th Intl conference on software engineering, ICSE '14, pp 676–687
9. De Lemos R, et al. (2013) Software engineering for self-adaptive systems: a second research roadmap. In: Software engineering for self-adaptive systems II. *LNCS*, vol 7475, pp 1–32. Springer
10. Dulac-Arnold G, Evans R, Sunehag P, Coppin B (2015) Reinforcement learning in large discrete action spaces. CoRR arXiv:1512.07679
11. Dutreilh X, Kirgizov S, Melekhova O, Malenfant J, Rivierre N, Truck I (2011) Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow. In: 7th Intl conference on autonomic and autonomous systems, ICAS'11, pp 67–74
12. Esfahani N, Elkhodary A, Malek S (2013) A learning-based framework for engineering feature-oriented self-adaptive software systems. IEEE Trans Softw Eng 39(11):1467–1493
13. Filho RVR, Porter B (2017) Defining emergent software using continuous self-assembly, perception, and learning. TAAS 12(3):16:1–16:25
14. Galindo JA, Benavides D, Trinidad P, Gutiérrez-Fernández AM, Ruiz-Cortés A (2019) Automated analysis of feature models: Quo vadis? Computing 101(5):387–433
15. Ghezzi C (2017) Of software and change. J Softw Evolut Process 29(9)
16. Hinchey M, Park S, Schmid K (2012) Building dynamic software product lines. IEEE Comput 45(10):22–26
17. de la Iglesia DG, Weyns D (2015) MAPE-K formal templates to rigorously design behaviors for self-adaptive systems. TAAS 10(3):15:1–15:31
18. Kaelbling LP, Littman ML, Moore AW (1996) Reinforcement learning: a survey. J Artif Intell Res 4:237–285
19. Kephart JO, Chess DM (2003) The vision of autonomic computing. IEEE Comput 36(1):41–50
20. Kinneer C, Coker Z, Wang J, Garlan D, Le Goues C (2018) Managing uncertainty in self-adaptive systems with plan reuse and stochastic search. In: 13th intl symposium on software engineering for adaptive and self-managing systems, SEAMS'18, pp 40–50
21. Mann ZÁ (2016) Interplay of virtual machine selection and virtual machine placement. In: 5th European conference on service-oriented and cloud computing, ESOCC'16, vol 9846, pp 137–151
22. Mann ZÁ (2018) Resource optimization across the cloud stack. IEEE Trans Parallel Distrib Syst 29(1):169–182
23. Metzger A, Bayer A, Doyle D, Sharifloo AM, Pohl K, Wessling F (2016) Coordinated run-time adaptation of variability-intensive systems: an application in cloud computing. In: 1st intl workshop on variability and complexity in software design, VACE@ICSE 2016, pp 5–11. ACM
24. Metzger A, Di Nitto E (2012) Addressing highly dynamic changes in service-oriented systems: towards agile evolution and adaptation. In: Agile and lean service-oriented development: foundations, theory and practice, pp 33–46

25. Metzger A, Pohl K (2014) Software product line engineering and variability management: achievements and challenges. In: Future of software engineering, FOSE'14, pp 70–84
26. Metzger A, Quinton C, Mann ZÁ, Baresi L, Pohl K (2020) Feature model-guided online reinforcement learning for self-adaptive services. In: Intl conference on service-oriented computing (ICSOC 2020), LNCS, vol 12571, pp 269–286. Springer
27. Moustafa A, Ito T (2018) A deep reinforcement learning approach for large-scale service composition. In: International conference on principles and practice of multi-agent systems, pp 296–311
28. Moustafa A, Zhang M (2014) Learning efficient compositions for QoS-aware service provisioning. In: IEEE intl conference on web services, ICWS'14, pp 185–192
29. Nachum O, Norouzi M, Xu K, Schuurmans D (2017) Bridging the gap between value and policy based reinforcement learning. In: Advances in neural information processing systems 12 (NIPS 2017), pp 2772–2782
30. Palm A, Metzger A, Pohl K (2020) Online reinforcement learning for self-adaptive information systems. In: Yu E, Dustdar S (eds) Int'l conference on advanced information systems engineering, CAiSE'20
31. Plappert M, Houthooft R, Dhariwal P, Sidor S, Chen RY, Chen X, Asfour T, Abbeel P, Andrychowicz M (2018) Parameter space noise for exploration. In: 6th intl conference on learning representations, ICLR 2018. OpenReview.net
32. Ramirez AJ, Cheng BHC, McKinley PK, Beckmann BE (2010) Automatically generating adaptive logic to balance non-functional tradeoffs during reconfiguration. In: 7th international conference on autonomic computing, ICAC'10, pp 225–234
33. Restuccia F, Melodia T (2020) DeepWiERL: Bringing deep reinforcement learning to the internet of self-adaptive things. In: IEEE INFOCOM 2020: IEEE conference on computer communications, pp 844–853. IEEE
34. Salehie M, Tahvildari L (2009) Self-adaptive software: Landscape and research challenges. TAAS 4(2)
35. Shaw R, Howley E, Barrett E (2021) Applying reinforcement learning towards automating energy efficient virtual machine consolidation in cloud data centers. Inf Syst.
36. Siegmund N, Kolesnikov SS, Kästner C, Apel S, Batory D, Rosenmüller M, Saake G (2012) Predicting performance via automated feature–interaction detection. In: 34th international conference on software engineering, ICSE'12, pp 167–177
37. Sutton RS, Barto AG (2018) Reinforcement learning: an introduction, 2nd edn. MIT Press, Cambridge
38. Taylor ME, Stone P (2009) Transfer learning for reinforcement learning domains: a survey. J Mach Learn Res 10:1633–1685
39. Thüm T, Batory D, Kastner C (2009) Reasoning about edits to feature models. In: 31st Intl conference on software engineering, ICSE'09, pp 254–264
40. Thüm T, Kästner C, Erdweg S, Siegmund N (2011) Abstract features in feature modeling. In: 15th international conference on software product lines, SPLC'11, pp 191–200
41. Van Der Donckt J, Weyns D, Quin F, Van Der Donckt J, Michiels S (2020) Applying deep learning to reduce large adaptation spaces of self-adaptive systems with multiple types of goals. In: 15th international symposium on software engineering for adaptive and self-managing systems, SEAMS 2020. ACM
42. Wang H, Gu M, Yu Q, Fei H, Li J, Tao Y (2017) Large-scale and adaptive service composition using deep reinforcement learning. In: 15th intl conference on service-oriented computing (ICSOC'17), pp 383–391
43. Wang H, Gu M, Yu Q, Tao Y, Li J, Fei H, Yan J, Zhao W, Hong T (2019) Adaptive and large-scale service composition based on deep reinforcement learning. Knowl-Based Syst 180:75–90
44. Weyns D (2021) Introduction to self-adaptive systems: a contemporary software engineering perspective. Wiley
45. Weyns D, et al (2013) Perpetual assurances for self-adaptive systems. In: de Lemos R, Garlan D, Ghezzi C, Giese H (eds) Software engineering for self-adaptive systems III. Assurances—International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers, *Lecture Notes in Computer Science*, vol 9640, pp 31–63. Springer
46. Zhao T, Zhang W, Zhao H, Jin Z (2017) A reinforcement learning-based framework for the generation and evolution of adaptation rules. In: Intl conference on autonomic computing, ICAC, pp 103–112