Yang WH, Pan MX, Zhou Y *et al.* Meaningful update and repair of Markov decision processes for self-adaptive systems. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 37(1): 106–127 Jan. 2022. DOI 10.1007/s11390-021-1484-8

Meaningful Update and Repair of Markov Decision Processes for Self-Adaptive Systems

Wen-Hua Yang[1,2,3] (杨文华), *Member, CCF*, Min-Xue Pan[2] (潘敏学), *Member, CCF*
Yu Zhou[1,2] (周　宇), *Senior Member, CCF*, and Zhi-Qiu Huang[1] (黄志球), *Senior Member, CCF*

[1] *College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China*
[2] *State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China*
[3] *Collaborative Innovation Center of Novel Software Technology and Industrialization*
Nanjing University, Nanjing 210023, China

E-mail: ywh@nuaa.edu.cn; mxp@nju.edu.cn; {zhouyu, zqhuang}@nuaa.edu.cn

Received March 31, 2021; accepted October 19, 2021.

Abstract Self-adaptive systems are able to adjust their behaviour in response to environmental condition changes and are widely deployed as Internetwares. Considered as a promising way to handle the ever-growing complexity of software systems, they have seen an increasing level of interest and are covering a variety of applications, e.g., autonomous car systems and adaptive network systems. Many approaches for the construction of self-adaptive systems have been developed, and probabilistic models, such as Markov decision processes (MDPs), are one of the favoured. However, the majority of them do not deal with the problems of the underlying MDP being obsolete under new environments or unsatisfactory to the given properties. This results in the generated policies from such MDP failing to guide the self-adaptive system to run correctly and meet goals. In this article, we propose a systematic approach to updating an obsolete MDP by exploring new states and transitions and removing obsolete ones, and repairing an unsatisfactory MDP by adjusting its structure in a more meaningful way rather than arbitrarily changing the transition probabilities to values not in line with reality. Experimental results show that the MDPs updated and repaired by our approach are more competent in guiding the self-adaptive systems' correct running compared with the original ones.

Keywords self-adaptive system, Markov decision process, model repair

1 Introduction

Modern software systems are growing at an astonishing rate in terms of scale, and many of them have close and prevalent interactions with the environment through the abundant equipped sensors and actuators [1]. Typical examples include autonomous vehicles, robotics, smart rooms, and Internet of Things, etc. The development of such complex software systems while considering the interactions can be laborious, not to mention the need to guarantee system qualities. One promising way to reduce effort is self-adaptation. Self-adaptive systems are generally considered to be efficient approaches for engineering resilient software systems in a cost-effective manner [2]. They are able to automatically adapt to the environmental condition changes in pursuit of user requirements. In general, a self-adaptive system is composed of the adaptation logic and the managed elements [3]. As the crux of a self-adaptive system, the adaptation logic is responsible for determining how to perform adaptation in complex environments.

To implement a concrete adaptation logic, many approaches have been proposed [4], e.g., rule-based [5,6], control theory based [7–9] and model-based approaches [10–13]. These approaches rely on certain explicit models, such as specified rules or probabilistic mod-

Regular Paper
Special Section on Software Systems 2021—Theme: Internetware and Beyond
This work was partially supported by the National Natural Science Foundation of China under Grant Nos. 61802179, 61972193 and 61972197, the Fundamental Research Funds for the Central Universities of China under Grant No. NS2021069, and the Natural Science Foundation of Jiangsu Province of China under Grant No. BK20201292.

els of system behaviours, to construct the adaptation logic. Among them, the probabilistic model, including discrete-time Markov chains (DTMCs) and Markov decision processes (MDPs), is one of the most widely-used approaches for building adaptation logics. DTMCs can model the probabilistic behaviours of self-adaptive systems, and MDPs extend DTMCs by allowing nondeterministic choices, which often exist in real-world self-adaptive systems.

Based on the MDP of a self-adaptive system, it is possible to generate a policy to control the system's behaviour. There is a large body of work focusing on generating optimal policies for self-adaptive systems modelled by MDPs[2, 13–15], and much progress has been made in this area. A common assumption of existing work that employs MDPs to build adaptation logics for self-adaptive systems is that once the MDPs are given, they are unlikely to change in the future. However, in reality, there are scenarios where the MDPs are subject to change. On the one hand, every MDP of a self-adaptive system is designed to deal with certain specific environmental conditions. When the system's operating environment becomes different from the ones assumed by the developers at the design time, the MDP will be obsolete, incapable of producing the possible policies that can be executed in the new environment, or containing policies that cannot happen in the new environment. On the other hand, when the self-adaptive system's MDP fails to satisfy properties, the model will become unsatisfactory and thus cannot provide policies that meet the goals. When such problems are encountered, it is better to update and repair the model if possible, rather than to redesign a new model from scratch.

In contrast to the policy generation, existing literatures have not thoroughly studied how to update and repair the underlying MDP models when necessary. An undeniable fact is that if relying on the obsolete and unsatisfactory MDP to generate policies, the policies obviously cannot guide the systems to run correctly. Hence, to address the above two problems, in this paper we propose a novel approach to updating and repairing obsolete and unsatisfactory MDPs. Specifically, when the environment changes, our approach identifies and removes the obsolete states and transitions, and finds new states and transitions that are feasible under the new environments. The repair of an unsatisfactory MDP is challenging since there can be numerous ways to change the MDP in terms of parameter and structure. Existing studies[16, 17] have identified the model repair problem and proposed to parametrically repair the MDPs, which is to change the transition probabilities. The repaired MDPs obtained from such approaches are compliant in the sense that they can satisfy the given properties. However, we recognise a problem with the parametric repair, i.e., the repaired models are not valid in the context of self-adaptive systems: the transition probabilities in MDPs for self-adaptive systems have practical meanings. They come from probabilistic system characteristics (e.g., a flawed actuator has a probability of executing undesired actions), and therefore, cannot be changed arbitrarily to values not in line with reality. Otherwise, the policy generated based on the repaired model is meaningless and cannot guide the system to run correctly. To address this problem, we propose a structural repair approach which finds alternative paths that can make the properties satisfied while keeping the transition probabilities consistent with reality. To efficiently construct the necessary structural repair, we novelly propose to leverage the parametric repair as an intermediate step and use its repair results to guide the structural repair of models so that the obtained models can satisfy the properties without violation.

In summary, this article makes the following contributions.

• Our approach addresses a novel problem of updating obsolete MDPs and repairing unsatisfactory MDPs for self-adaptive systems.

• Our approach structurally updates and repairs MDPs to cope with environment changes and obtain compliant and valid repair results.

• For two representative self-adaptive systems, a total of 12 MDPs are experimented for evaluation. The results show that our approach is effective and outperforms existing approaches.

The rest of the article is organised as follows. Section 2 provides some background for self-adaptive systems and a motivating example for approach demonstration purposes. Section 3 presents the details of our approach. The evaluation is presented in Section 4. Section 5 discusses the threats to validity. Related work is discussed in Section 6, and finally Section 7 concludes the article.

2 Background and Motivating Example

In this section, we first introduce the basic concepts for self-adaptive systems, particularly their modelling with MDPs, and then we motivate our work using an autonomous car system example.

2.1 Self-Adaptive System Modelling

A self-adaptive system is capable of adjusting its behaviour in response to the changes of the environmental condition. Typically, such a system's operation can be described as the following process. It monitors the state of the environment and the system by sensors, and analyses whether an adaptation is required or not to achieve the desired goals. When necessary, it devises a plan and executes the action specified in the plan through its actuators. Communication between different adaptation phases is conducted through a shared knowledge-base. This process forms the well-known MAPE-K loop [18], a concept model for self-adaptive systems.

When it comes to implementing the adaptation logic for self-adaptive systems, existing approaches vary from simple rule-based algorithms [5, 6] to complex approaches based on, e.g., learning techniques [19], goal models [20], control theory [7–9] and probabilistic models [12, 13, 21, 22]. Among them, probabilistic models, especially DTMCs and MDPs, have been widely applied because of their rich expressiveness and available formal guarantees. DTMCs are well suited for modelling probabilities, and MDPs extend DTMCs by allowing nondeterministic choices that are common in self-adaptive systems. Since self-adaptive systems need to make adaptation decisions based on the sensed environment and system states, it is necessary to support predicates, instead of atomic propositions in the classic model, to specify whether certain decision criteria are satisfied. In this article, we adopt an extended version of the classic MDP, supported by PRISM [23], which allows predicates in the states and guards associated with the actions.

Definition 1. *An MDP is a tuple* $\mathcal{M} = (S, s_0, A, \tau, \mathcal{L}, \mathcal{R})$, *where*

- *S is a finite non-empty set of states, and $s_0 \in S$ is the initial state*;
- *A is a finite non-empty set of actions, where each $a \in A$ is associated with a guard ψ which is a set of predicates* (*taken from a set $\mathcal{P}$*), *and the set of actions enabled from state s is denoted by A_s*;
- *$\tau : S \times A \times S \to [0, 1]$ is a transition probability function such that for $s \in S$, $a \in A$, either $\sum_{s' \in S} \tau(s, a, s') = 1$* (*$a$ is enabled*) *or $\sum_{s' \in S} \tau(s, a, s') = 0$* (*$a$ is disabled*), *for each $s \in S$ there exists at least one action enabled from s*;
- *$\mathcal{L} : S \to 2^{\mathcal{P}}$ is a labelling function assigning to each state to a subset of predicates* (*taken from a set $\mathcal{P}$*) *that are true in that state*;
- *$\mathcal{R} : S \times A \to \mathbb{R}_{\geqslant 0}$ is a reward structure. If an action a is chosen in state s, a reward of $r(s, a)$ is obtained.*

An MDP describes how the state of a system can evolve in discrete time steps. In each state s, the choice of which action to take from A_s is nondeterministic. We can use a policy to resolve the nondeterministic choices. A policy of an MDP is a function π that specifies the action $\pi(s)$ to be chosen in state s. The guard of an action a is denoted as $a.\psi$, and once a is selected in state s and its guard $a.\psi$ is satisfied, then a transition to a successor state s' occurs randomly according to the transition probability function $\tau(s, a, s')$, and the predicate $\mathcal{L}(s')$ in s' holds. Depending on the types of predicates, e.g., propositions or predicates in arithmetic, the MDP can model many properties that need to be satisfied by the system. A policy controls the execution of the MDP, and each execution is a path which is a (finite or infinite) sequence of transitions $\sigma = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots$ through the MDP such that when s_i transits to s_{i+1} by performing a_i, the guard $a_i.\psi_i$ is satisfied, and the predicate $\mathcal{L}(s_{i+1})$ evaluates to true. How to obtain a suitable policy is an important problem, and existing studies [2, 15, 21, 22] have proposed many methods to check for the existence of policies or optimise policies regarding some given quality goals. However, few researchers have studied how to update MDPs for self-adaptive systems when the environment has changed or repair MDPs when they cannot meet the quality goals. Therefore, this article focuses on updating and repairing MDPs rather than generating policies, which is a practical need as illustrated by the following motivating example.

2.2 Motivating Example

Autonomous car systems perceive their surroundings through sensors and make movements based on the sensed environment. When designing such a system, developers would consider possible environmental conditions that the car may encounter or summarise the environmental conditions that the car has encountered from its running and testing history. Then, they design corresponding adaptation logics about how to react to those environmental conditions. Generally, developers are incapable of designing an adaptation logic that can cope with all environmental conditions. Instead, developers would design different logics for pre-defined subset of environmental conditions.

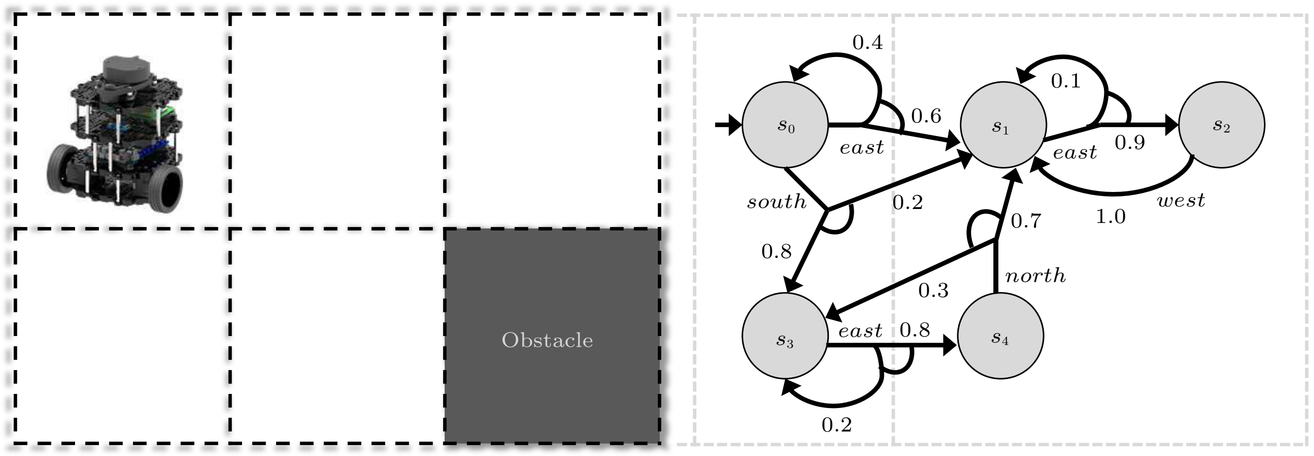
Fig. 1 shows a running scenario of an autonomous car system and an MDP $\mathcal{M}$ designed for this system.

The car is equipped with distance measuring sensors to sense the distances to its surrounding obstacles. The system controls the car to explore the area as extensive as possible while not bumping into any obstacles. The design of $\mathcal{M}$ adopts a common paradigm from existing studies[14, 15], i.e., the abstraction of states is conducted by discretizing the space into equidistant and non-overlapping regions. $\mathcal{M}$ has five states, each of which represents a grid cell of the terrain divided into 3×2 grid cells (except for the one occupied by the obstacle). In every state, one or more actions from the set $A = \{east, west, south, north\}$ are available, and make the car move between grid cells. Due to the uncertainty, e.g., flawed actuators, the car could probabilistically move to an alternative state when executing a certain action. Taking state s_0 in Fig.1(b) for example, by executing action *south* the car could transit to s_1 or s_3 at a probability of 0.2 and 0.8, respectively.

Based on $\mathcal{M}$, we can generate policies (depending on the context, also known as strategies, adversaries or schedulers) in many ways to act as the autonomous car system's adaptation logic. Compared with the much attention gained by the policy generation, the update of MDP is less mentioned. Yet it is a very practical but over-neglected need. The motive for updating a self-adaptive system's MDP comes from multiple facets. First, the system's operating environment becomes different from the ones assumed by the developers at the design time. For example, the obstacle in the second row in Fig. 1(a) could be moved to its western grid cell. In that case, the original MDP $\mathcal{M}$ is obviously not suitable for the system, since the grid cell of state s_4 is occupied by a new obstacle and state s_3 should not execute action *east* to transit to s_4. Meanwhile, the grid cell previously occupied by the obstacle is now empty and can be reached from adjacent grid cells. We need to update $\mathcal{M}$, including the states and transitions, to specify these possibilities. Furthermore, the predicates in the model also need to be updated. The predicate, taken from $\mathcal{P}$ and assigned to a state by $\mathcal{L}$, reflects the system's runtime conditions and should be true in its corresponding state. The predicates associated with the actions are used to form guards that test whether the action can be executed. Let us suppose that s_3's predicate $\mathcal{P}_3$ is $eastern > 1$ that requires the distance to the eastern obstacle larger than 1 and s_4's predicate $\mathcal{P}_4$ is $eastern > 0$ that requires the distance to the eastern obstacle larger than 0. These predicates of different states indicate the system's distances to the obstacles. Then, the guard of action *east* (moving for 1 unit) should be $\mathcal{P}_3 \wedge (eastern - 1 \rightarrow \mathcal{P}_4)$, to ensure that when *east* is taken, the system can transit from s_3 to s_4. Most of the predicates are related to the environmental conditions, and some states' predicates need updating as the environment changes, especially for new states. For example, if the obstacle moves to its northern grid cell, the predicate of s_3 should be updated. Second, the designed $\mathcal{M}$ does not satisfy a given property ϕ. Self-adaptive systems are often used in safety-critical scenarios, and if the property cannot be satisfied, it is desirable to find $\mathcal{M}'$ that satisfies ϕ at a low cost.

3 Updating and Repairing Approach

In this section, we will detail our approach for updating and repairing MDPs for self-adaptive systems,



(a) (b)

Fig.1. A simplified autonomous car system and its MDP. (a) Running scenario. (b) MDP designed for the system.

starting with an approach overview.

3.1 Approach Overview

Our approach deals with the model obsolete and model unsatisfactory problems, as shown in Fig.2. The environment of a self-adaptive system can constantly be changing, and it is possible that under the new environment, the MDP is incapable of specifying the possible policies that can be executed, or contains policies that cannot happen, which is considered as obsolete. To update such an obsolete model, our approach first performs a structural adjustment. Infeasible states and related transitions are removed according to the environment changes, and new states and related transitions are added when necessary. Meanwhile, predicates and transition probabilities of affected states and transitions are updated to keep up with the changed environment, as explained in Subsection 3.2.

An MDP is unsatisfactory if it cannot generate any feasible policies with respect to the given goals. Repairing an unsatisfactory model can be challenging if the result needs to be both compliant (having feasible policies) and valid (reflecting real system characteristics). Existing studies[16, 17] have proposed several techniques to repair MDPs. However, their goal is to find an $\mathcal{M}'$ that satisfies ϕ and differs from $\mathcal{M}$ only in the transition probability, which is compliant but not necessarily valid to our problem. Since the transition probabilities of $\mathcal{M}$ reflect self-adaptive system's characteristics (e.g., flawed actuators), we cannot change them to the values that are not in line with reality; otherwise, the generated policies from $\mathcal{M}'$ are meaningless and cannot guide the system's correct operation. Subsection 3.3 will present a well-designed algorithm that identifies the compliant and valid changes by leveraging the results of existing model repair techniques (i.e., the parametric repair, cf. Subsection 3.3.1), to achieve our meaningful repair purpose.

3.2 Obsolete Model Update

When the environment changes more than expected, in the new environment there could emerge some new environmental conditions that cannot be handled by the MDP. The direct impact of environment changes on an MDP goes on the state abstraction and transitions. It will make the abstract states unsuitable to represent the actual new environment and the transitions unable to execute. By removing infeasible states and transitions and exploring new states and transitions, we adjust the structure of an MDP to make it accordant with the new environment. As the states and transitions are removed and added, the predicates and probabilities also need updates.

Generally, to model a self-adaptive system, we need to be aware of the system logic, the system status, the environment, and the state abstraction approach. The environment and system status can be very complex. Fortunately, a self-adaptive system only uses its sensed results to make decisions, and therefore we only need to consider part of the environment and the system status. The process of modelling can be summarised as follows. The part of environment and the system status, which need to be taken into the modelling process, can be specified as mapping relations $E : V_E \rightarrow U_E$ and $C : V_C \rightarrow U_C$, respectively, where V_E and V_C are sets representing all the attributes of the environment and the system of interest, respectively, and U_E and U_C are sets of concrete values. E and C are not part of the MDP model; however, as the self-adaptive systems determine their states by sensing, the states of the model

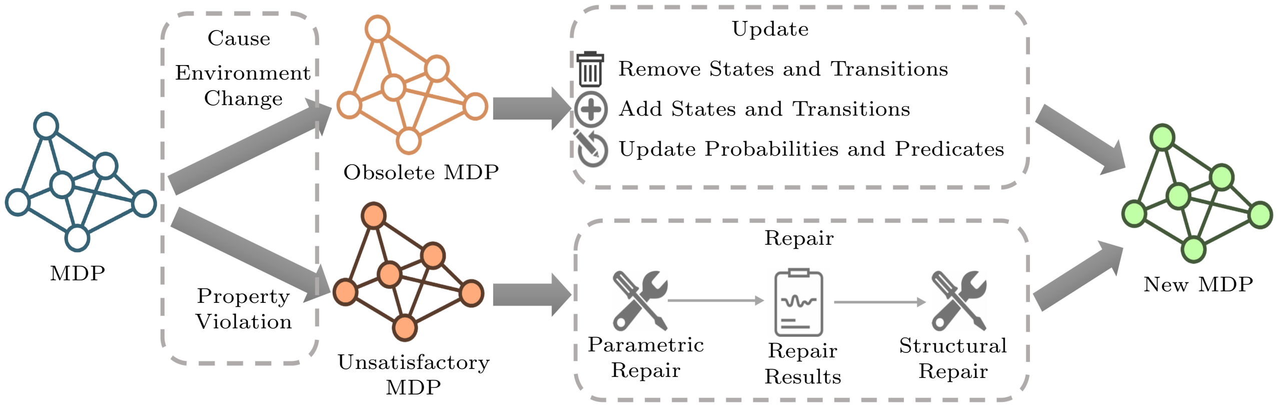


Fig.2. Overview of our updating and repairing approach.

are encoded from E and C. Let $\{E\}$ and $\{C\}$ be the sets of all the variable-value mappings defined by E and C, respectively. Then the abstraction of state, denoted as *Abs*, is a function $2^{\{E\}} \times 2^{\{C\}} \to S \cup \mu$ that encodes the part of the environment and the system of interest, represented by subsets of $\{E\}$ and $\{C\}$, to a state in S or a null state μ that does not appear in the model, and there is $\mathcal{L}(s)$ for each $s \neq \mu,\ s \in S$. The encoding of states from the environment and system status is different application by application. However, for all applications, once a model is obtained, *Abs* maintains the relation among the environment and the system status, and the states. Taking the autonomous car system as an example, environment E can specify the area for exploration (e.g., length $\times$ width for a rectangle area) and the positions of obstacles (e.g., specified by coordinates). Then the set of states S can be encoded from E as the grid cells, splitting the area, that the car can go in. Reversely, we can also know from each state $s \in S$ its position in the area and that it is not occupied by an obstacle. Moreover, we can know from $\mathcal{L}(s)$ the distance of the grid cell to the obstacles.

The system logic is modelled by probabilistic transitions between states such that $s_i \xrightarrow{a} s_j$, $\tau(s_i, a, s_j) = p$, where s_i and s_j are states in S, a is an action, and p is a probability. The logic specifies that when the system is in state s_i, it is allowed to perform action a if guard $a.\psi$ is satisfied, and transfer to state s_j at a probability of p (in the following, we will omit the probabilities unless they are required for computation). The guard of action a needs to ensure that when a is taken, the predicates in s_j need to be satisfied. In the autonomous car system example, action a can be action *east*, *west*, *south* or *north*. These actions' guards, for example, can be, when moving towards a direction, the distance to the obstacles are still larger than a certain value.

The modelling of self-adaptive systems has been studied by many researchers and is not the subject under study in this article. Nevertheless, we can learn from the above introduction that when the environment E changes, the states in the model should be updated accordingly. However, the problem of how to update the MDP model has not been well-studied, and in the following three subsections, we will present our solution.

3.2.1 Removing States and Transitions

For an MDP $\mathcal{M}$, when the environment changes, we need to find the states and transitions that are affected by the change of the environment. There are two kinds of effects on states. First, an existing state $s \in S$ can become unreachable in the new environment, which requires to eliminate s from S. Second, there may be a new system state s_{new} that has not been modelled but exists under the new environment, which requires to add s_{new} to S.

The environment change manifests in the mapping relation $E : V_E \to U_E$ such that the values for the environment attributes can be different in the new environment. Let the old environment be E and the new environment be E', where $E = V \to U$ and $E' = V \to U'$. For any $(v_i \to u_i) \in \{E\}$, if v_i is not changed in the environment, then $(v_i \to u_i) \in \{E'\}$; otherwise, $(v_i \to u_i') \in \{E'\}$, where u_i' is the new value representing the environment change. Given an MDP $\mathcal{M}$ and the new environment E', to identify the unreachable states we perform a reachability check for the states in $\mathcal{M}$ under E'. Specifically, for each $s' \in S$, we first find all $s \in S$ and $a \in A$ such that $\tau(s, a, s') \neq 0$. Then we check if $\tau(s, a, s')$ is still feasible by checking the satisfaction of the following two conditions under E': 1) s and s' are valid under E', i.e., there are $e, e' \in 2^{E'}$ and $c \in 2^C$ such that $Abs(e, c) = s$ and $Abs(e', c) = s'$; and 2) when $\mathcal{L}(s)$ is satisfied and $a.\psi = \text{true}$, then if a is taken, $\mathcal{L}(s')$ is satisfied. If $\tau(s, a, s')$ is infeasible, i.e., the above conditions are not both satisfied, we eliminate this transition by setting $\tau(s, a, s')$ to 0. For state s', if all $\tau_{s \in S, a \in A}(s, a, s')$ are 0, then we remove state s' and $\tau_{s \in S, a \in A}(s, a, s')$. In this way, the unreachable states and infeasible transitions will be removed.

3.2.2 Adding States and Transitions

Changes in the environment may also result in new feasible system states. In this article, we explore the new states and associated transitions based on the existing MDP. After the environment is changed to a new one E', for each $s \in S$, we try every action $a \in A$ on s to check whether any new state will emerge under the new environment E', i.e., $\exists s_{\text{new}} \notin S, s \xrightarrow{a} s_{\text{new}}$. The identification of a new state depends on the state abstraction approach: using the current environment condition $e \in 2^{\{E\}}$ and the system condition $c \in 2^{\{C\}}$ obtained by executing a in state s, we can get the current state cs through the abstraction function, i.e., $Abs(e, c) = cs$. However, the design of the abstraction function, varying from application to application, is beyond the scope of this article. When a new state s_{new} that has not been modelled is identified, we add it to S. Meanwhile, $\tau(s, a, s_{\text{new}})$ will be added to the model. We also need to check whether there are transitions from s_{new} to other states (or itself). To find feasible

$\tau_{s' \in S, a \in A}(s_{\text{new}}, a, s')$, we conduct each possible $a \in A$ in s_{new} to check which state s' it can transit to. If we find $\mathcal{M}$ can transit from s_{new} to s' by executing a, we add transition $\tau(s_{\text{new}}, a, s')$ to $\mathcal{M}$.

Our approach of exploring new states and transitions starts from the existing MDP, which may leave out the states that are not reachable from the existing ones in the MDP. We propose this approach based on the following reasons. On the one hand, we believe that the existing MDP contains the main logic of the system and should be reused as much as possible, and thus we focus on updating and repairing it. On the other hand, exploring the states that are reachable from existing ones is more efficient and practical than systematically exploring states in the entire state space.

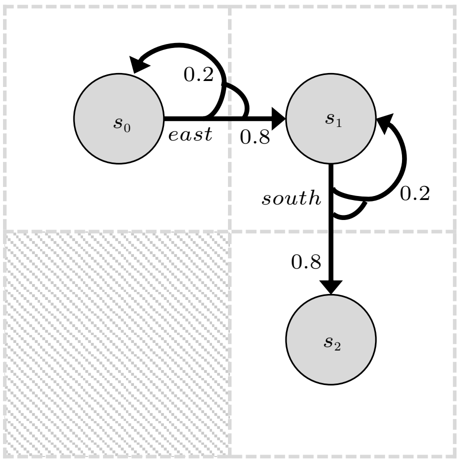
3.2.3 *Updating Probabilities and Predicates*

The above structural change of an MDP $\mathcal{M}$ will remove/add states and transitions. Accordingly, the probabilities of the affected transitions and the predicates over the affected states and actions also need to be updated. Generally, for any state $s \in S$, if a transition $\tau(s, a, s'), s' \in S$ is added or removed during the model structural change, the probabilities of any transition starting from s with action a need to be updated adhering to the rules as follows: 1) the sum of the probabilities on all the transitions starting from s with action a should equal 1, i.e., $\sum_{s' \in S} \tau(s, a, s') = 1$; 2) the probability of each transition $\tau(s, a, s')$ should realistically reflect the causes for nondeterminism. Such causes, which manifest in the transition probabilities, may be the imperfection of actuators and sensors, or the choice of system logic design, etc. These manifestations are application-specific and should be treated differently. Nevertheless, as our approach is based on an existing model $\mathcal{M}$, it is possible to leverage the probabilities in $\mathcal{M}$ to update the affected transitions. Still suppose that a state $s \in S$ has a transition with action a added or removed. It is often the case that for all the current transitions starting from s with action a, we can find another state s^* such that each $\tau(s, a, s'), s' \in S$ can be mapped to $\tau(s^*, a, s'^*), s'^* \in S$. Then the probabilities of the transitions starting from s can be updated proportionally to the probabilities of their mapped transitions starting from s^*. This update process is exemplified in Subsection 3.2.4. There is a chance that the affected transitions cannot be mapped to others, and in this case, it means the types of these transitions are new to the model and their probabilities need to be given by the designer. Considering that, in practice, most environmental changes are modest, the update of transition probabilities can often be done automatically.

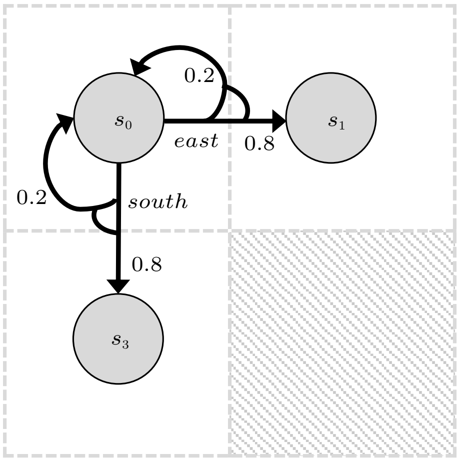
The main function of predicates is to decide whether an action can be taken at the current state s. Action a can be taken when the predicate $\mathcal{L}(s)$ of s is satisfied, and predicate $a.\psi$ can be satisfied, i.e., $\mathcal{L}(s) \wedge a.\psi$. Therefore, we update the predicates from the point of view of actions. For every removed transition $\tau(s, a, s')$, we check whether there is any $s' \in S$ such that $\tau(s, a, s') \neq 0$. If not, the guard $a.\psi$ will also be removed from the predicate of s. For example, in the autonomous car system, the execution of the action *east* needs to satisfy a precondition $\textit{eastern} > 1$ requiring that there should be no obstacle in the eastern of the car within 1 unit of distance. When all the transitions with such an action are removed, the state which is the source of these transitions no longer needs the predicate $\textit{eastern} > 1$. Similarly, for every added $\tau(s, a, s')$, the predicate over s and s' can be updated by adding the guard condition concerning action a.

3.2.4 *Updating Example*

Using the autonomous car system, we will give simplified examples of removing/adding states, transitions, and updating probabilities and predicates, respectively, to illustrate the approach of updating obsolete models. Fig.3 shows how the MDP is updated when the environment changes, i.e., the obstacle moving from the left-bottom to the right-bottom. The state abstraction *Abs* is defined by discretising the space into equidistant and non-overlapping regions. Since in the new environment, the obstacle moves to the grid cell of s_2, making s_2 no longer valid, and s_2 is removed. Meanwhile, the transition $\tau(s_1, \textit{south}, s_2)$ becomes infeasible and thus is removed. When exploring new states and transitions, a new state s_3 is added by trying feasible action *south* in s_0. From state s_1 in the existing MDP, we know that when taking the action *south* on a state, it can transit to the source state again ($\tau(s_1, \textit{south}, s_1)$) or to the state representing its southern grid cell ($\tau(s_1, \textit{south}, s_2)$). These two transitions can be mapped to the newly-added transitions $\tau(s_0, \textit{south}, s_0)$ and $\tau(s_0, \textit{south}, s_3)$ of the new MDP, which enables us to set the probabilities of the new transitions to 0.2 and 0.8, respectively. The predicate over state s_1 should also be updated since $\tau(s_1, \textit{south}, s_2)$ is removed. For example, it does not need to require the distance to the southern obstacle larger than 10 units to safely conduct action *south*.



(a)



(b)

Fig.3. Simplified example for updating an MDP. (a) Model before the update. (b) Model after the update.

3.3 Structural Model Repair

Given an MDP $\mathcal{M}$ and a property ϕ specified in PCTL (probabilistic computation tree logic)[24] to be satisfied, we employ PRISM[23], a widely-used probabilistic model checker, to encode and verify the MDP $\mathcal{M}$. PCTL is an extension of computation tree logic that allows for probabilistic quantification of described properties, and is commonly used to express probabilistic properties of Markov models. It includes a probabilistic operator P to express probabilistic properties. Consider a PCTL formula of the form $P_{\bowtie p}[\varphi]$, where φ is the path formula occurring inside the probabilistic operator P, $\bowtie \in \{<, \leqslant, >, \geqslant\}$, and $p \in [0,1]$ is a probability threshold. A PCTL formula $P_{\bowtie p}[\varphi]$ can be used to specify whether the probability that φ is satisfied over paths meets the threshold requirement ($\bowtie p$). PRISM is a probabilistic model checker, a tool for formal modelling and analysis of systems that exhibit random or probabilistic behaviour. PRISM has been successfully used to analyse systems from many different application domains. It provides an easy-to-use language to encode the models to be verified and supports a wide range of quantitative properties. PRISM incorporates state-of-the-art symbolic data structures and algorithms and also includes a discrete-event simulation engine, providing support for approximate/statistical model checking, and implementations of various different analysis techniques, such as quantitative abstraction refinement and symmetry reduction. There are many existing studies[21,22] that have used PRISM to verify MDP-based self-adaptive systems and the detailed tutorials and examples on how to encode and verify MDP against PCTL properties by PRISM①.

When the verification results show that $\mathcal{M}$ cannot satisfy ϕ, it means that the policy generation approaches cannot find any policy satisfying ϕ in $\mathcal{M}$. Instead of designing a new MDP from scratch, it is more cost-effective to repair the existing model $\mathcal{M}$ to make ϕ satisfied. Researchers have identified this model repair problem for probabilistic systems (e.g., Markov chains and MDPs)[16,17]. However, they treated the repair just as a change in the transition probabilities in their work, and therefore, intended to find an $\mathcal{M}'$ satisfying ϕ that just differs from $\mathcal{M}$ in the transition probability. We call this kind of repair as parametric repair. However, the probabilities' values in MDPs of self-adaptive systems cannot be changed arbitrarily. Without considering the causes or meanings of the probabilities, parametric repair does not guarantee to produce valid repair for real-world self-adaptive systems. Therefore, their repair approach cannot be directly adapted to our problem. Exploiting the repair results of parametric repair, we propose an algorithm to repair an unsatisfactory MDP in a more meaningful way by adjusting the structure without changing the transition probability arbitrarily, as explained in Subsections 3.3.1 and 3.3.2.

3.3.1 Parametric Repair

We start with a brief introduction to the parametric repair approach[16,17]. The approach is based on a parametric MDP, which is defined as a tuple $\tilde{\mathcal{M}} = (S, s_0, A, \tau + \boldsymbol{Z}, \mathcal{L}, \mathcal{R}_a)$ where $S, s_0, A, \tau, \mathcal{R}_a$ are defined as in Definition 1, and $\boldsymbol{Z}$ is $S \times A \times S \rightarrow span(K)$ where $K = \{x_1, \cdots, x_n\}$ and $span(K) = \{\omega_1 x_1 + \cdots +$

①PRISM. https://www.prismmodelchecker.org/, May 2021.

$\omega_n x_n | \vec{\omega} \in \mathbb{R}^n_{>0}\} \subset \mathcal{F}_K$ ($\mathcal{F}_K$ denotes the set of rational functions from K to $\mathbb{R}$). Given a parametric MDP and an evaluation $k : K \to \mathbb{R}$, we can induce an MDP $\mathcal{M}_k \overset{\text{def}}{=} (S, s_0, A, \tau_k, \mathcal{L}, \mathcal{R}_a)$ where for s, $s' \in S$, $a \in A$, $\tau(s, a, s') \overset{\text{def}}{=} \tau(s, a, s') \langle k \rangle$. The parametric repair is to find an evaluation k which satisfies $k \in Evals(\mathcal{M})$ and $\mathcal{M}_k \vDash \phi$. The set of effective evaluations $Evals(\mathcal{M})$ is defined as the set of all evaluations k such that $\mathcal{M}_k$ is a well-defined MDP (cf. Definition 1) and for all $s \in S$, $a \in A_s$ and $s' \in S$ we have that either $\tau(s, a, s') \langle k \rangle \neq 0$ or $\tau(s, a, s') \langle k' \rangle = 0$ for all evaluations k'. To our problem, supposing we are going to repair an MDP $\mathcal{M}$ using parametric repair, we can first construct a parametric MDP $\tilde{\mathcal{M}}$ whose elements are the same as $\mathcal{M}$ except adding $\boldsymbol{Z}$ and then find the effective evaluation k which represents the changes on transition probabilities.

We adopt the sampling-based approach introduced in [17] to solve the parametric repair problem. The sampling-based approach involves a randomised search through the parameter space, efficiently yielding some good parameter values whose weighted distance from the original one is sufficiently small (the weights can be chosen to express the importance or priority of certain parameters over the others) rather than finding all valid values or the closest value. Monte Carlo sampling techniques are applied to the model repair problem. This kind of repair does not change the structure of the MDP.

3.3.2 *Structural Repair Algorithm*

The parametric repair aims to make $\mathcal{M}$ satisfy ϕ by changing the transition probabilities. The results of the parametric repair can be represented as a series of changes $(s_i, a_m, s_j) \to pr_n$, which means the original transition probability $\tau(s_i, a_m, s_j)$ should be changed to pr_n. To avoid directly modifying these probabilities values, we propose to explore new transitions and combine them with the ones in the model, to achieve an equivalent effect of changing probabilities. In a nutshell, instead of changing $\tau(s_i, a_m, s_j)$, we try to build an alternative path from s_i to s_j such that the multiplication of the transitions' probabilities in that path equals (or approximately equals) $\tau(s_i, a_m, s_j)$.

Algorithm 1 shows the process of the structural repair. The inputs are an MDP $\mathcal{M}$, the results of parametric repair $C = \{(s_{i_1}, a_{m_1}, s_{j_1}) \to pr_{n_1}, \cdots, (s_{i_t}, a_{m_t}, s_{j_t}) \to pr_{n_t}\}$ for $\mathcal{M}$, and a given threshold ϵ to decide whether two probabilities are close enough. For each $(s_{i_q}, a_{m_q}, s_{j_q}) \to pr_{n_q}$ in C, we start from s_{i_q} (line 2) and repeatedly extend the path in a depth-first manner (line 4) until a path that ends with s_{j_q} having the expected probability ($|probability(path) - pr_{n_q}| < \epsilon$) is found (lines 9–11), or no such path exists (line 3). The function `extendPrefix` extends the path by enumerating all transitions that can be appended to the current path stored in variable *path* and can make the new one a prefix of the alternative path for $\tau(s_i, a_m, s_j)$, i.e., it can be extended to a path that starts from s_i and ends with s_j. If all transitions have been tried and no such path is found (line 5), the algorithm performs a backward operation by removing the last transition in *path* (`removeLastTransition`, line 6). Then new transitions are explored by applying feasible actions on the last state in *path* (`generateNewTransiton`, line 7). Recall that the states encode a self-adaptive system's understanding to the environment. Therefore, action a is feasible w.r.t. state s, if guard $a.\psi$ is satisfied in state s, and after a is taken and the model transits to s', $\mathcal{L}(s')$ holds. These newly-obtained transitions are enumerated to extend *path*. When all the parametric repair results in C have been processed, Algorithm 1 returns the repaired MDP $\mathcal{M}'$. Algorithm 1 may fail to return a repaired MDP even if an effective evaluation is found for the MDP during the parametric repair. However, other evaluation results from the parametric repair may yield successful structural repair. Therefore, we can repeat the above process to repair the MDP. A termination criterion can be set to stop the repeating process, such as a successful repair obtained or the process being repeated a given number of times.

Algorithm 1. Structural Repair Algorithm

```
Input: an MDP $\mathcal{M}$, the parametric repair's result $C$, and
       a given threshold $\epsilon$
Output: the repaired MDP $\mathcal{M}'$
1  foreach $(s_{i_q}, a_{m_q}, s_{j_q}) \to pr_{n_q}$ do
2      $path \leftarrow s_{i_q}$;
3      while $path! = \text{null}$ do
4          $path \leftarrow \text{extendPrefix}(path)$;
5          if $!isExtended$ then
6              $path \leftarrow \text{removeLastTransition}(path)$;
7              $\text{generateNewTransition}(path.lastState)$;
8          if $isPath(path)$ then
9              if $|probability(path) - pr_{n_q}| < \epsilon$ then
10                 break;
11             else
12                 $path \leftarrow \text{removeLastTransition}(path)$;
13 return $\mathcal{M}'$;
```

3.3.3 Repairing Example

We present a simplified example in Fig. 4 to illustrate how structural repair works. Fig. 4(a) shows a part of an MDP for an autonomous car system. In this model, states s_3 and s_4 are not directly connected to s_0, s_1 or s_2, but to some other states, as represented by the dashed lines. Suppose the property to be satisfied by this MDP is that the probability of the car not colliding with any obstacle is greater than 0.9 ($P_{>0.9}$ [$\mathbf{G}$ $\neg$*bump*]), and for this MDP the result of parametric repair is $C = \{(s_0, a_1, s_1) \to 0.4, (s_0, a_1, s_2) \to 0.6\}$ (i.e., red edges in Fig. 4(a)). To avoid directly changing the values of $\tau(s_0, a_1, s_1)$ and $\tau(s_0, a_1, s_2)$, the structural repair algorithm explores new transitions and constructs two alternative paths $\sigma_1 = s_0 \xrightarrow{a_3} s_4 \xrightarrow{a_5} s_1$ and $\sigma_2 = s_0 \xrightarrow{a_2} s_3 \xrightarrow{a_4} s_2$ to replace the parametric repair's result. The newly explored transitions are the red directed edges in Fig. 4(b). The probabilities of σ_1 and σ_2 are $0.5 \times 0.8 = 0.4$ and $1.0 \times 0.6 = 0.6$, which are just the values required for $\tau(s_0, a_1, s_1)$ and $\tau(s_0, a_1, s_2)$, respectively.

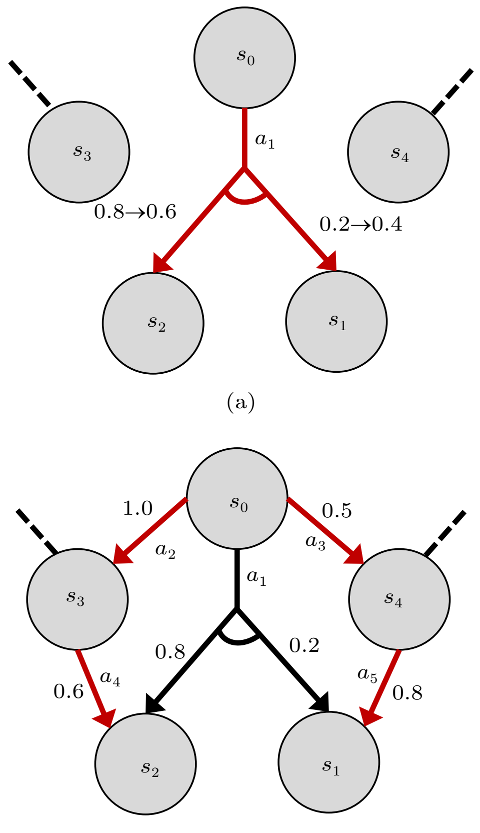


Fig.4. An illustrative example for MDP repair. (a) The model before repairing. (b) The model after repairing.

4 Evaluation

In this section, we evaluate the effectiveness of our approach, and investigate the following research questions.

- *RQ*1. Can our updating and repairing approach effectively improve the adaptability of self-adaptive systems?
- *RQ*2. How does the structural repair perform compared with the parametric repair for self-adaptive systems?
- *RQ*3. How efficient is our approach in updating and repairing MDPs?
- *RQ*4. Is our updating and repairing approach more cost-effective while maintaining correctness as compared with building the MDP from scratch?

4.1 Experimental Setup

We implemented our approach based on PRISM (Version 4.4-osx64), and the experimental environment was an Intel Core i7 3.2 GHz and 32 GB RAM machine. To select adequate experimental subjects, we examined existing related work and a site containing a set of exemplars for self-adaptive systems[2]. Two typical self-adaptive systems — the autonomous car system [15] and an exemplar for self-adaptive Internet of things (DeltaIoT) [25] were selected as our subjects. The autonomous car system's operating scenario has been introduced in Subsection 2.2, and here we briefly explain the DeltaIoT system. It consists of a multi-hop network comprising multiple motes distributed in various buildings of the campus. Motes are placed to provide access control to labs, monitor the occupancy status and sense the temperature via various sensors. The sensor data from all the buildings are relayed to the IoT gateway, and there is a management tier connecting to the gateway via the Internet. Each mote must have a path towards the gateway along with other motes, and meanwhile the IoT system is expected to consume less energy while offering reliable communication. To achieve the goal, the system needs to adapt the network settings of the IoT motes (e.g., transmission power and spreading factor).

For the autonomous car system, we collected seven MDPs derived from [14, 15] and their references. Since DeltaIoT is designed for evaluating adaptive solutions, it does not provide any adaptation logic (e.g., MDP models). We designed five MDPs to control DeltaIoT's adaptation for different scenarios. The states in

[2]https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/, Sept. 2021.

DeltaIoT represent the states of all the motes and the actions include adapting the transmission power, modifying the path to the gateway and the spreading factor settings, and adding or removing links, etc.. Environment changes could be mote lost or mote addition. The probability comes from the success/failure rates of the execution of actions and network transmission.

The autonomous car system can run in a real environment Turtlebot[3] or in a simulator, where MDPs can be used as adaptation logics. DeltaIoT provides an open-source and customisable simulator for experiments, and we customised the simulator for experiments. These systems need to satisfy predefined properties during the operation. Table 1 presents the probabilistic properties that need to be satisfied for each of the 12 different self-adaptive systems. The first seven rows are the properties for the autonomous car systems, and the last five rows are the properties for DeltaIoT. These properties are all probabilistic ones. For example, the first property $P_{<0.1}[\varphi]$ requires that the probability of the car bumping into any obstacle and failing to traverse to the area's four corners is at most 0.1.

4.2 RQ1: Overall Effectiveness

The adaptability of a self-adaptive system can be considered as the ability to adjust its behaviour in response to environmental condition changes. The more adaptable a self-adaptive system is, the more likely it runs correctly. The adaptation logic is responsible for determining how to perform adaptation. For self-adaptive systems modelled with MDPs, their adaptation logics are specified in the model. The 12 MDPs of the autonomous car system and DeltaIoT are designed to handle some specific environments. If we change the environment of an MDP to a different one that is not assumed by the designers, the old MDP will be obsolete and is unlikely to be able to deal with the new environment. If an MDP does not satisfy a given property, we consider this MDP unsatisfactory to the property. To assess the effectiveness and practicality of our update and repair approach, we compared the running of self-adaptive systems guided by the original MDPs, which are obsolete and unsatisfactory, and by the new MDPs obtained by applying the update and/or repair approaches on the original ones.

Specifically, we devised environment changes in the autonomous car system and DeltaIoT (e.g., moving obstacles and adding and losing motes). These environment changes would lead to obsolete models. For example, an obstacle would restrict the car's movement around the obstacle, making the original state corresponding to the grid cell occupied by the obstacle inaccessible. However, removing an obstacle can lead to new transitions between states corresponding to the grid cells occupied by and adjacent to the obstacle. It is similar for the DeltaIoT system that its model would become obsolete when motes are added or lost. As the model becomes obsolete, some properties would be violated, and we collected those properties for each of the

Table 1. Properties That Need to Be Satisfied for Each of the 12 Experimental Subjects

Subject	Property	Description
Car-1	$P_{<0.1}[\varphi]$	The probability of the car hitting any obstacle and failing to traverse to the area's four corners is at most 0.1
Car-2	$P_{>0.9}[\varphi]$	The probability that the car does not collide with any obstacle is greater than 0.9
Car-3	$P_{>0.8}[\varphi]$	The probability of the car not hitting any obstacle and traversing to the area's four corners is greater than 0.8
Car-4	$P_{<0.05}[\varphi]$	The maximum probability of the car failing to traverse to the area's four corners is at most 0.05
Car-5	$P_{>0.9}[\varphi]$	The probability that the car does not collide with any obstacle is greater than 0.9
Car-6	$P_{>0.8}[\varphi]$	The probability of the car not hitting any obstacle and traversing to the area's four corners is greater than 0.8
Car-7	$P_{<0.1}[\varphi]$	The probability of the car hitting any obstacle and failing to traverse to the area's four corners is at most 0.1
DeltaIoT-1	$P_{<0.05}[\varphi]$	The maximum probability of losing any message is at most 0.05
DeltaIoT-2	$P_{>0.85}[\varphi]$	The probability of successful transmission of any message with low energy consumption is greater than 0.85
DeltaIoT-3	$P_{<0.2}[\varphi]$	The maximum probability of losing any message with low energy consumption is at most 0.2
DeltaIoT-4	$P_{>0.95}[\varphi]$	The probability of successful transmission of any message is greater than 0.95
DeltaIoT-5	$P_{>0.7}[\varphi]$	The probability of successful transmission of any message with high energy consumption is greater than 0.7

[3]https://www.turtlebot.com, Sept. 2021.

12 MDPs. Then we applied our approach to update and/or repair these MDPs (during repair, the parametric repair was set to be run just once). The update and the repair were both conducted automatically. For the two examples, the state abstraction function *Abs* can be specified by a set of rules. For example, the state abstraction of the autonomous car system is specified by rules concerning the splitting of the area to be grid cells, the positions of the obstacles and the distances to the obstacles of each grid cell. Additionally, the affected transitions were all mapped to other transitions in the existing model, from which their probabilities were updated proportionally using the approach in Subsection 3.2.3. Depending on the decision about whether we update or repair the models, we will have 48 MDPs falling into four categories (each category has 12 MDPs), as shown in Table 2. The four categories of models are closely related: models in category 1 are the original ones and models in the other categories are obtained by updating or repairing models in category 1.

Table 2. Four Categories of MDPs

Category	Updated	Repaired
1	No	No
2	Yes	No
3	No	Yes
4	Yes	Yes

We generated the policies using classical approaches[26] for the 48 MDPs and then employed them to guide the running of the autonomous car systems and DeltaIoT under the new changed environments. For the autonomous car system, we conducted the experiments both by real cars (Turtlebot) and in the simulator, and for DeltaIoT we carried out the experiments in the open-source simulator. During the systems' running, we recorded whether the systems were running correctly and whether they satisfied the given properties.

On the other hand, in addition to using MDPs to construct the adaptation logic for self-adaptive systems, other approaches such as rule-based or model-based approaches can also be employed. However, it is uncertain whether using these approaches to construct self-adaptive systems can cope with environmental changes, especially when the system's operating environment becomes different from what the developers expected during the design phase. Therefore, to further validate the necessity and effectiveness of our approach, we compared it with existing rule-based and model-based approaches. We chose two commonly-used approaches for constructing self-adaptive systems for comparisons. The first approach is based on a model of adaptive behaviour called adaptation finite-state machine (A-FSM)[6, 27], and the second approach[28, 29] proposes an interactive state machine (ISM) to construct self-adaptive systems. Similar to the above experiments, we first designed the corresponding adaptation logic for the above 12 systems in their original environments using these two approaches, and then deployed and tested the adaptation logics. Afterwards, we observed whether these adaptation logics could still guide the correct operation of the systems in the newly changed environments and whether their properties could be satisfied.

Results. If an error occurs in the system, we consider that the system is not running correctly. For our subjects, the error could be the car bumping into obstacles or the data of motes not being relayed to the IoT gateway. Each of the 48 MDPs, the 12 A-FSMs and the 12 ISMs was run in the simulator for 20 times to check if it was running correctly. Besides, each of the seven autonomous car systems was also executed in the field for 20 times. Fig.5 shows the occurrence numbers of errors in the simulator for the four categories of MDPs, A-FSMs, and ISMs. As we can see from Fig.5, both A-FSMs and ISMs performed poorly in the new environments, because, similar to MDPs, they can only be designed to take into account some environmental conditions that the system may encounter. Therefore, when the environment changes, A-FSMs and ISMs also face the problem of model obsolescence, making the corresponding adaptation logic unable to guide the system to operate correctly and thus produce errors. This further demonstrates the necessity for our updating and repairing approach, i.e., it is difficult to avoid the problem addressed in this article even if other rule-based or model-based approaches are employed to construct adaptive logic for self-adaptive systems.

Under the new environment, the MDPs in category 1 and category 3 encountered much more errors than the MDPs in the other two categories, since they were not updated for the environment changes. Particularly, MDPs in category 3 did not show a reduced number of errors even they had been repaired. Without the model update, the repair was conducted on the premise of an inaccurate environment. Therefore, the fewer errors of MDPs in category 2 and category 4 show that our model update approach is effective in dealing with environmental changes and thus can reduce errors. Note that it is possible that even if the model precisely reflects

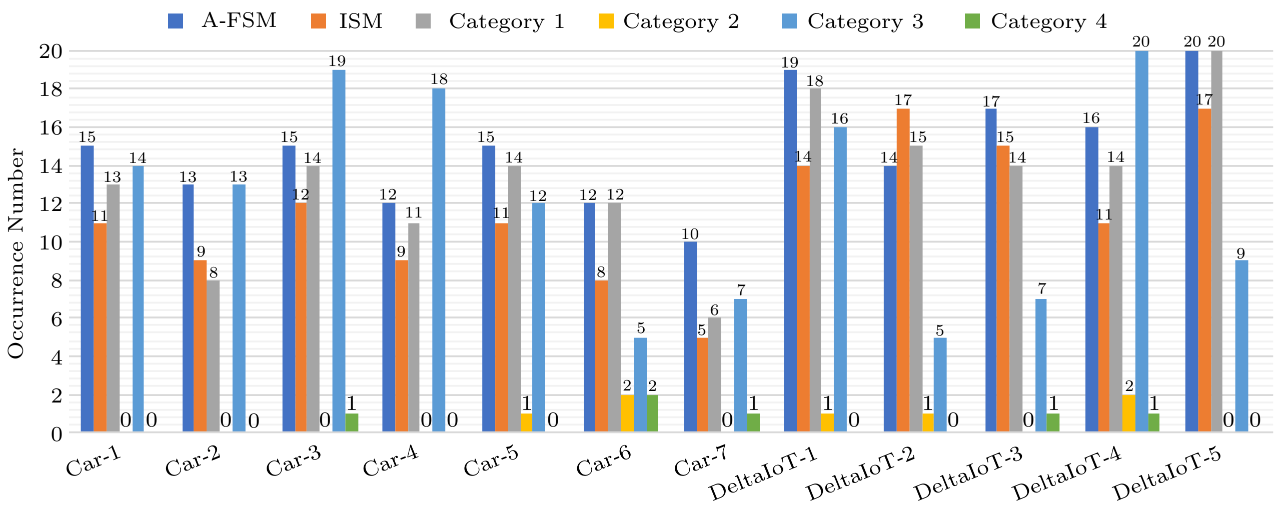


Fig.5. Occurrence number of errors in the simulator for experimental subjects.

the system and the environment, the actual behaviours of the system are probabilistic and some of them may manifest in errors. The result is also consistent with the data from the field study of the autonomous car systems, which is shown in Table 3. This further confirms that obsolete models cannot guide the system' correct running, and the model update is necessary for self-adaptive systems.

For the properties, the judgement of whether they are satisfied during the systems' running is relatively complex since the properties are often related to probabilities. For example, in DeltaIoT, there is a property requiring that the successful transmission of a message is eventually witnessed with a probability greater than or equal to 0.9 ($P_{\geqslant 0.9}$ [**F** "*trans_msg*"]). For these probabilistic properties, we cannot draw the conclusion based on a single run of the system, but rather on the probability estimated from multiple runs. Therefore, to obtain an accurate estimation of probabilities, we ran each of the 48 MDPs, 12 A-FSMs, and 12 ISMs in the simulator for 1 000 times to provide more sampling data. We assigned the same set of probabilistic properties to each of the A-FSMs, ISMs and the four categories, in which each of the models has a probabilistic property. The property is in the form of requiring the probability of a specific event to occur to be larger or less than a specified value. We recorded the occurrence numbers of the required event for each model in its running of 1 000 times to estimate the probability of the occurrence of the event.

Since the properties are related to the probability, to further mitigate the randomness effect on the judgement of probabilistic properties, we ran the Wilcoxon signed-rank test [30] with Bonferroni correction [31] to check whether the differences between our approach (category 4) and other alternatives in satisfying the probabilistic properties are statistically significant. We consider that one approach performs significantly better in satisfying properties than the other one at the confidence level of 95% if the corresponding Wilcoxon signed-rank test result (i.e., p-value) is less than 0.05.

Table 4 and Table 5 show the results for the autonomous car systems and the DeltaIoT systems, respectively. In the two tables, for properties that are required to be less than a value, the smaller probabilities obtained from experiments are more inclined to pro-

Table 3. Occurrence Number of Errors in Reality for the Seven Car Systems Controlled by Different Models

Model	Car-1	Car-2	Car-3	Car-4	Car-5	Car-6	Car-7
A-FSM	15	16	19	16	14	12	11
ISM	13	9	16	14	12	9	6
Category 1	18	12	20	15	15	13	10
Category 2	1	1	0	0	0	0	0
Category 3	15	17	18	18	8	9	13
Category 4	0	0	0	0	0	0	0

Table 4. Property Violations for the Four Categories of MDPs, A-FSMs, and ISMs of the Car

Model	Car-1	Car-2	Car-3	Car-4	Car-5	Car-6	Car-7	Testing
	($p < 0.1$)	($p > 0.9$)	($p > 0.8$)	($p < 0.05$)	($p > 0.9$)	($p > 0.8$)	($p < 0.1$)	(p-Value)
A-FSM	0.232	0.353	0.196	0.292	0.435	0.310	0.206	$< 10^{-3}$
ISM	0.206	0.368	0.219	0.304	0.458	0.385	0.212	$< 10^{-3}$
Category 1	0.221	0.339	0.201	0.312	0.408	0.210	0.287	$< 10^{-3}$
Category 2	0.164	0.654	0.446	0.244	0.660	0.430	0.272	$< 10^{-3}$
Category 3	0.190	0.403	0.398	0.259	0.529	0.302	0.191	$< 10^{-3}$
Category 4	0.045	0.910	0.783	0.037	0.967	0.902	0.068	–

Table 5. Property Violations for the Four Categories of MDPs, A-FSMs, and ISMs of DeltaIoT

Model	DeltaIoT-1	DeltaIoT-2	DeltaIoT-3	DeltaIoT-4	DeltaIoT-5	Testing
	($p < 0.05$)	($p > 0.85$)	($p < 0.2$)	($p > 0.95$)	($p > 0.7$)	(p-Value)
A-FSM	0.182	0.659	0.414	0.370	0.468	$< 10^{-3}$
ISM	0.195	0.675	0.402	0.392	0.471	$< 10^{-3}$
Category 1	0.204	0.628	0.469	0.383	0.435	$< 10^{-3}$
Category 2	0.194	0.647	0.309	0.403	0.794	$< 10^{-3}$
Category 3	0.140	0.668	0.336	0.414	0.543	$< 10^{-3}$
Category 4	0.061	0.924	0.126	0.991	0.894	–

perty satisfaction, while for those greater than a value, the larger probabilities are more favourable. From Table 4 and Table 5, we can see that neither A-FSM nor ISM performed well, much the same as category 1. For all the 12 systems, those A-FSMs and ISMs are unable to make the systems satisfy their respective properties after the change of the environment. This is because the adaptive logics originally constructed using A-FSMs and ISMs are also incapable of coping with the new environment. For the MDPs, we can see that model update alone can also be effective for probabilistic properties (category 2 outperforms category 1 for all the 12 MDPs), since errors caused by environment changes can result in the unsatisfaction of the properties. Meanwhile, the data in category 3 are all better than the data in category 1 too, showing the effectiveness of model repair. It is worth mentioning that the effect of applying model update alone (category 2) is better than that of just using model repair (category 3) for probabilistic properties in eight out of the 12 MDPs. The reason is that while model repair can resolve some problems of properties not meeting the probabilistic requirements, it cannot avoid the errors caused by environment changes. Therefore, the combination of both model update and repair is necessary, as confirmed by the good performance of category 4. To further confirm this, we conducted hypothesis tests (Wilcoxon signed-rank tests) on the results of all the other approaches against the results of category 4 to determine whether category 4 outperforms the other approaches in terms of satisfying properties. The last column in Table 4 and Table 5 are the results of the testing, and it can be seen that all p-values are much less than 0.05, which means that category 4 is significantly better than the other approaches in satisfying the properties.

4.3 RQ2: Comparison of Repair Approaches

In the repair of an unsatisfactory MDP, the parametric repair approach will try to revise the transition probabilities to make the model satisfy the given properties. In Subsection 3.3, we proposed a structural repair approach to repair the MDPs for self-adaptive systems, and in this experiment we compared the results of MDPs, repaired by parametric repair and structural repair approaches, guiding the running of self-adaptive systems. Specifically, we implemented the parametric repair approach and our approach, and applied them to repair the MDPs of the autonomous car systems and the DeltaIoT (without changing their environments) that do not satisfy the given properties, respectively. Then, based on the repaired MDPs of the two approaches, we generated policies and used them to guide the systems' operation in the simulators for 1 000 times. The running results were recorded to check whether the properties were satisfied by the repaired MDPs during the systems' running. Besides, we also ran the Wilcoxon signed-rank test with Bonferroni correction to check whether our structural repair performs significantly better than other approaches.

Results. Table 6 and Table 7 show the experimental results, which are values representing the actual

Table 6. Property Violations of MDPs Repaired by Different Approaches for the Car

Approach	Car-1 ($p < 0.1$)	Car-2 ($p > 0.9$)	Car-3 ($p > 0.8$)	Car-4 ($p < 0.05$)	Car-5 ($p > 0.9$)	Car-6 ($p > 0.8$)	Car-7 ($p < 0.1$)	Testing (p-Value)
`Original`	0.290	0.362	0.440	0.238	0.492	0.354	0.196	$< 10^{-3}$
`Parametric`	0.472	0.109	0.211	0.764	0.090	0.243	0.475	$< 10^{-3}$
`Structural`	0.037	0.922	0.812	0.042	0.945	0.884	0.073	–

Table 7. Property Violations of MDPs Repaired by Different Approaches for DeltaIoT

Approach	DeltaIoT-1 ($p < 0.05$)	DeltaIoT-2 ($p > 0.85$)	DeltaIoT-3 ($p < 0.2$)	DeltaIoT-4 ($p > 0.95$)	DeltaIoT-5 ($p > 0.7$)	Testing (p-Value)
`Original`	0.185	0.557	0.259	0.506	0.545	$< 10^{-3}$
`Parametric`	0.389	0.372	0.682	0.288	0.086	$< 10^{-3}$
`Structural`	0.048	0.936	0.109	0.922	0.896	–

probabilities of property satisfaction. The probabilities were calculated by dividing the number of the system runs that satisfies the properties by the number of total runs (1 000 times). Row `Original` shows the results of using unrepaired MDPs, and row `Parametric` and row `Structural` show the results of using MDPs repaired by the parametric repair approach and our structural repair, respectively. As we can see from the two tables, the probabilities in row `Parametric` have larger gaps to the expected ones (row `Property`) than the probabilities in row `Original`, suggesting that using parametrically repaired MDPs are even worse than not using any repair techniques. The reason, as we investigated, is that almost all the transition probabilities generated by the parametric repair are inconsistent with the actual probabilities of the systems. As a result, the policies generated from the parametrically repaired MDPs often lead the system to violate them rather than to guide the system to meet the properties. Our structural repair, on the contrary, does not change the transition probabilities but looks for alternative paths with the expected probabilities. As Table 6 and Table 7 show, using structural repair can make the originally unsatisfied properties become satisfied for 11 out of 12 MDPs (except for DeltaIoT-4 in which the structural repair also makes improvement). Considering the hypothesis testing, the last columns of Table 6 and Table 7 show the testing results for MDPs repaired by our structural repair approach versus the original MDPs and MDPs repaired by the parametric repair approach, respectively, and the p-values are also much less than 0.05, indicating that our structural repair performs significantly better than the others. In summary, the results of using structural repair are consistently better than those of using the parametric repair or not using any repair techniques for all the 12 MDPs.

4.4 RQ3: Approach Efficiency

Efficiency is an important evaluation aspect of an approach. Our approach can update and repair MDPs for self-adaptive systems. Therefore, we also investigated how efficient our approach is in terms of updating and repairing MDPs. To answer RQ3, we applied our updating and repairing approach to the 12 MDPs of the autonomous car systems and DeltaIoT. Since the updating and repairing can be applied separately or together as needed, it is necessary to know how much time and memory they consume, respectively. We first applied the updating approach and the repairing approach to the 12 MDPs, respectively. Then we used them together and recorded the time and memory cost for these three scenarios.

Results. The time and the memory cost for updating and/or repairing the MDPs are shown in Table 8 for the car systems and in Table 9 for the DeltaIoT systems. For the autonomous car systems, the update of an MDP took 15 seconds on average, the repair took 95 seconds on average, and the two combined took 123.6 seconds on average. The repair needs more time than the update since it has to conduct a parametric repair first and may need to enumerate many paths, while the update operations are just in proportion to the number of states in the MDP. Besides, we can see that the time cost for combining update and repair was slightly more than the sum of the time to apply the update and the repair separately. The reason is that when combining the update and the repair, the update was applied first and can result in more states and transitions than the original MDP, causing the repair to take more time.

Table 8. Time and Memory Cost of Our Approach for the Car

System	Update		Repair		Combined	
	Time (s)	Memory (MB)	Time (s)	Memory (MB)	Time (s)	Memory (MB)
Car-1	17.0	33.0	86.0	147.0	134.0	152.0
Car-2	8.0	55.0	26.0	170.0	30.0	182.0
Car-3	33.0	97.0	90.0	211.0	115.0	271.0
Car-4	9.0	56.0	143.0	337.0	155.0	410.0
Car-5	5.0	46.0	212.0	290.0	239.0	321.0
Car-6	12.0	48.0	31.0	195.0	50.0	180.0
Car-7	21.0	104.0	77.0	168.0	142.0	199.0
Average	15.0	62.7	95.0	219.9	123.6	245.0

Table 9. Time and Memory Cost of Our Approach for DeltaIoT

System	Update		Repair		Combined	
	Time (s)	Memory (MB)	Time (s)	Memory (MB)	Time (s)	Memory (MB)
DeltaIoT-1	34.0	201.0	432.0	453.0	665.0	776.0
DeltaIoT-2	78.0	189.0	963.0	1 012.0	1205.0	1 448.0
DeltaIoT-3	44.0	355.0	743.0	707.0	906.0	951.0
DeltaIoT-4	90.0	409.0	942.0	1 206.0	1 139.0	1 519.0
DeltaIoT-5	37.0	220.0	325.0	509.0	389.0	871.0
Average	56.6	274.8	681.0	777.4	860.8	1 113.0

For the memory cost, the repair also took more memory than the update. However, when combining the two together, the memory cost can be less than the sum of that of applying update and repair separately (in nine out of 12 MDPs). The reason is that even in the combined approach, the update and the repair are applied in different stages, and thus the repair, which is applied after the update, may reuse some of the memory consumed in the first stage.

For the DeltaIoT systems, when comparing the costs of update, repair and their combined approach, the same results are witnessed. Furthermore, compared with the autonomous car systems, the time and memory costs of DeltaIoT are relatively higher. This is reasonable because the MDPs of DeltaIoT contain more states and transitions and are thus more complex, as shown in Table 10. On average, the time cost is 860.8 seconds and the memory cost is 1 113 MB when the update and the repair are applied together. Such a cost is acceptable, considering that the update and the repair are mostly conducted offline. It is known that a predefined MDP of a self-adaptive system can cope with certain dynamic changes in runtime. However, not all changes are manageable: drastic changes in the system's operating environment can make the MDP fail to satisfy the property. An offline operation would be induced to address this situation, which can be a redesign of the model, or alternatively, a more cost-effective update and repair.

4.5 RQ4: Cost-Effectiveness

When the self-adaptive system's MDP model becomes obsolete or does not satisfy the properties, besides updating and repairing the system's MDP using our approach, another option is to design a new MDP model from scratch. To assess which approach is more cost-effective while ensuring that the property is satisfied, we designed a controlled experiment with in total 14 graduate students (as experiment subjects) majoring in computer science for evaluation. The controlled

Table 10. Size of the MDPs Used for Experiments

Element	Car-1	Car-2	Car-3	Car-4	Car-5	Car-6	Car-7	DeltaIoT-1	DeltaIoT-2	DeltaIoT-3	DeltaIoT-4	DeltaIoT-5
States	20	12	32	12	18	24	19	58	40	66	52	73
Trans.	41	37	76	38	40	55	30	104	88	109	93	115

experiment aims to evaluate the cost-effectiveness via comparing the MDP updated and repaired by our approach and the MDP redesigned by the subjects. We assessed the cost-effectiveness in two aspects: 1) the time required to update and repair, or redesign the MDP; and 2) whether the new MDP satisfies the given property.

All the participants were enrolled in either the M.S. or Ph.D. programs and had taken at least one software modelling course. Furthermore, all the participants were asked about their background on MDP before the experiment session. The results were used as a measure of the randomised process design to divide the subjects into groups to ensure that the background of MDP of each group is closely equivalent. Before the experiment was conducted, a one-hour lecture was given by the first author of the article to all the subjects about MDP. After the lecture, we also conducted a quiz to confirm that the subjects understood the fundamentals of MDP and its modelling.

The controlled experiment employed the seven autonomous car systems for evaluation, by considering that the case study is relatively simpler than DeltaIOT since we need to ensure that the subjects were able to understand the context of the case study within a limited period of time. For these seven systems, we devised environment changes and then used our approach to update and repair their original MDPs, while asking subjects to redesign the MDPs for these systems in the new environment. The subjects were divided into two groups, one with prior knowledge of the MDPs of the autonomous car systems from the training lecture and the other without. These two groups simulated realistic situations where the developers redesign the systems designed by themselves (group 1), and where the developers redesign systems designed by others (group 2), respectively.

Results. Table 11 shows the results for the controlled experiment. Both group 1 and group 2 have seven subjects. We randomly assigned seven autonomous car systems to the subjects in each group; therefore, each subject needed to redesign an MDP for one system. Then, we recorded and calculated the average time they took. Before the experiment, we told the subjects that we had a generous time limit, i.e., we requested them to understand the system and complete the redesign process within two hours. If they did not complete the redesign, they were counted as having exceeded the time limit, but in fact they all completed it within the time limit. We then ran each of these systems 1 000 times in the simulator based on their redesigned MDPs to determine if the systems' properties were satisfied. At the same time, we also used our approach to update and repair the MDPs of these systems, and then check in the simulator whether the processed MDPs satisfy the properties.

Table 11. Results of the Controlled Experiment

Group	Time	Number of Satisfied MDPs
1	1.1 (h)	4
2	1.5 (h)	2
Our approach	2.0 (min)	7

Note: The "time" in the table refers to the average time spent on redesigning or updating and repairing the MDPs, and "number of satisfied MDPs" indicates the number of MDPs that satisfy the given properties.

The average time used by the subjects and our approach, and the number of MDPs in the seven systems that do not satisfy the given properties for each group and our approach are given in Table 11. As we can see from Table 11, the time consumed by our approach is much less than the time used to redesign the MDPs by the subjects in group 1 and group 2. It is worth mentioning that our approach also outperforms the other two groups in terms of satisfying properties. It can be seen that among the seven systems, the MDPs updated and repaired by our approach satisfy all the given properties, while group 1 and group 2 satisfy only four and two properties out of seven properties, respectively. Since the subjects in group 1 know the MDPs of the autonomous car systems in advance, its result is slightly better than that of group 2, but still inferior to our approach. These results demonstrate that it is more cost-effective to use our approach to update and repair an existing MDP than to design a new MDP from scratch.

5 Threats to Validity

The main threat to construct validity concerns the simulators used to evaluate the approaches. Since the experiments for counting whether the probabilistic properties are violated need to run for lots of times, it is impractical and very time-consuming to conduct the experiments in the real world. However, the simulators may not fully reflect the actual running of the systems. To reduce this threat, we improved the simulator by repeatedly comparing the simulator with the actual execution of the system. Moreover, we also employed another open-source and well-known simulator to reduce implementation bias. In our experiments, we also resorted to hypothesis testing to help us further confirm the differences of satisfying probabilistic properties in the simulator between those approaches. Under this premise, we think this threat is reduced.

Threats to internal validity concern factors internal to our study that could have influenced the results. One possible threat is that not any property can be made satisfied through our MDP repair. Properties can be used to specify users' requirements and goals for the system, and they have different degrees of strength, e.g., the property "the probability of a failure occurring is at most 10^{-5}" is stronger than the property "the probability of a failure occurring is at most 10^{-2}". It is possible that a property is so strong that no policy exists to make it satisfied. Besides, properties specified incorrectly can also be impossible to satisfy. For such properties, the repair of MDPs will surely fail.

Threats to external validity concern the generalization of our conclusions, which may not generalize to other self-adaptive systems. To reduce this threat, we selected 12 different self-adaptive systems from the autonomous car and Internet of Things domain. Meanwhile, we made our best efforts in collecting diverse MDPs or having MDPs independently developed by different developers. The results consistently support our approach, and we try to make them applicable to other self-adaptive systems. Still, there is a need for evaluating our approach with more other systems.

6 Related Work

Self-adaptive systems have been a focus of research study due to their ability to adapt to changes, as introduced in the survey article about the landscape of research, taxonomies, gaps, and future challenges in self-adaptive systems [32]. In general, a self-adaptive system is composed of the adaptation logic and the managed elements [3]. As the crux of a self-adaptive system, the adaptation logic is responsible for determining how to perform adaptation in complex environments.

Different approaches are proposed for implementing a concrete adaptation logic. We can roughly categorise these approaches into rule-based, control theory based, and model-based approaches. Rule-based approaches use rules to model a system's reactions to monitored events [5, 6]. One obvious drawback of the rule-based approach is that the rules are hard to specify and may need to be updated frequently, which makes the system not flexible enough. Zhao [33] identified this problem and proposed a rule generation and evolution approach for requirements-driven self-adaptive systems. However, the generated rules are all in the form of "if-then" whose expression ability is limited and cannot avoid the possible conflict between rules, and the evolution process merely deals with the change of user goals. Some researchers have constructed self-adaptive systems that carry out the decision-making process by using control theory [7–9]. These studies mainly employ control theory to construct adaptation logic for functional or non-functional objectives, but they require the developers to fully understand the system and control theory to identify knobs, actuators, or provide abstraction specification, etc. Model-based approaches aim to provide correctness assurance for self-adaptive systems since they are often used in safety-critical scenarios, e.g., self-driving cars. In this category of work, probabilistic models (e.g., DTMCs and MDPs) have a dominating position. The implementation of the model-based approaches could be achieved under the Rainbow framework [34], which is one of the most well-known architectural frameworks for self-adaptive systems. Also based on architectural models, Chen *et al.* [35] proposed to use the models to support behavioural adaptation for self-adaptive systems. The architecture model is specified by the architecture description language Wright#, and a genetic algorithm based technique was proposed to search for behavioural alternatives. Business value is employed to evaluate the behavioural alternatives while capturing the trade-offs among relaxed functional and quality constraints. These model-based approaches [5, 6, 33–35] provide effective ways to construct the adaptation logic, while they pay less attention to updating and repairing the existing logic when it becomes obsolete and unsatisfactory. Note that MDP is also used in reinforcement learning, but the distinction needs to be drawn that reinforcement learning is different from our work. As a mathematical framework for modelling decision making, MDP was known at least as early as the 1950s [36]. Reinforcement learning is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximise the notion of cumulative reward [37]. The environment is typically stated in the form of an MDP. Thus, reinforcement learning uses an MDP to describe the process of learning where the probabilities or transitions are unknown. Differently, our work and other related work [2, 13–15] in the software engineering area are to use MDPs (as a model) to model the self-adaptive systems, that is, to specify the states, probabilities, and transitions for the self-adaptive systems. Meanwhile, the decision-making process of a self-adaptive system is also not a learning process.

Cámara and Lemons[10] proposed an approach that uses a DTMC to model the adaptation behaviour of the system to obtain levels of confidence regarding the resilience of each adaptation. In [11], the authors derived a DTMC from a formal architecture description of the managed system with the changes imposed by each strategy and used the model to optimise the self-adaptation decisions to fulfill the desired quality goals. Also based on DTMCs, Filieri and Tamburrelli[12] focused on probabilistic runtime model checking for self-adaptive systems. It belongs to the runtime quantitative verification technique, which can be used to support adaptation as discussed in [38]. Similar to us, [13] also employs MDPs to calculate which is the best path of execution regarding the system quality goals. In [39], an iterative decision-making scheme has been proposed for MDPs and applied to self-adaptive systems. Considering that some probability parameters in some MDPs may not be fixed, the scheme infers both point and interval estimates for the undetermined transition probabilities in an MDP based on sampled data, and iteratively computes a confidently optimal scheduler from a given finite subset of schedulers. Compared with DTMCs, MDPs extend DTMCs by allowing nondeterministic choices, which are closer to most real-world self-adaptive systems. Besides these related studies, [4] discusses more different approaches to engineering self-adaptive systems.

Most above-mentioned solutions of designing adaptation logic for self-adaptive systems either rely on the manually-defined specifications (e.g., rules) or generate policies/plans from models (e.g., policies generation from MDPs[13, 21, 22]). When the underlying models fail to support adaptation, instead of abandoning the models, an alternative is to update and repair the models. However, this update and repair issue has not been thoroughly studied, especially for MDPs. Sykes *et al.*[19] used a probabilistic rule learning technique to generate new adaptation plans for rule-based self-adaptive systems. Filieri *et al.*[40] proposed an approach to learning and updating DTMCs. Different from our work, they focused on learning and updating the transition probabilities of DTMCs instead of repairing the model, and they assumed the structure of DTMCs does not change. Therefore, their approaches cannot adapt to our problem for updating and repairing MDPs. To support dynamic software update for systems that require continuous operation, studies[41, 42] employ the controller synthesis to solve the dynamic controller update problem which describes how the system copes with specification changes in the requirements and/or the environment. In these studies, the controller is specified by the labelled transition system or finite state machine other than MDPs used in our work. Hence, their goals are not the same as ours, and their technology cannot be used to solve our problems.

Technically, our work is related to the model repair of probabilistic models (e.g., DTMCs and MDPs). An approach is introduced in [16] to repairing unsatisfactory probabilistic models based on parametric model checking. Afterwards, Chen *et al.*[17] proposed a solution for repairing MDPs. For these two studies, their goals are both to find a change in transition probabilities to make the model satisfy the given properties. This solution is not valid for real-world self-adaptive systems modelled with MDPs because the transition probabilities cannot be changed arbitrarily. Our approach is to modify the structure to achieve the meaningful repair. Not much work has been carried out to directly study MDPs' repair, while some related work exists regarding the parametric synthesis of MDPs[43–45], which is a complementary technique to model repair of MDPs. The main research object of these studies is parametric MDPs, in which some of the transition probabilities depend on a set of parameters. Parameter synthesis of MDPs is to find ranges of parameter values such that a satisfaction probability of a formula meets a given threshold, is maximised, or is minimised respectively. Although their goal is different from MDP repair, these techniques can be used in the future to further improve the effectiveness of the repair of MDPs. In addition to repairing MDPs, DTMCs' repairing has also been of interest to researchers. A greedy approach for repairing DTMCs has been proposed in [46]. To repair the DTMC, they iteratively consider single probability distributions in isolation, and modify the parameter values to decrease the probability to move to more dangerous successor states. An effort for presenting an approach for the repair of DTMCs is also discussed in [47]. In that work, the authors proposed a framework based on abstraction and refinement of DTMCs, which reduces the state space of the probabilistic system to repair at the price of obtaining an approximate solution. Still, these efforts focus on model repair by probability modification, which means that during model repair, the structure of the model is preserved, and only some of the transition probabilities of the Markov model are modified. Differently, our approach modifies the model's structure to achieve a meaningful repair of MDPs.

7 Conclusions

In this article, we identified two critical problems (i.e., model obsolete and model unsatisfactory) existing in self-adaptive systems modelled by MDPs. An obsolete MDP could be the result of the environmental change, which makes the environment different from the ones assumed by the developers. An MDP is considered unsatisfactory when it fails to satisfy the given properties. These two problems can cause the policies generated from such MDPs unable to guide systems to run correctly. To address them, we first updated the obsolete MDPs by performing a structural adjustment to remove and/or add states and the related transitions when the environment changes. Then we proposed a structural repair approach to repairing the unsatisfactory MDPs in a more meaningful way: different from existing model repair techniques that repair the MDP by modifying the transition probabilities, we adjusted the structure of the MDP to avoid arbitrarily changing transition probabilities to values not in line with reality.

We evaluated our approach on two kinds of representative self-adaptive systems, i.e., the autonomous car systems and the self-adaptive Internet of things systems (DeltaIoT). Experimental results showed that our updating and repairing approach can effectively update the obsolete MDPs used to model self-adaptive systems and repair those MDPs when they do not satisfy the given properties. Compared with the original MDPs that have not been updated and repaired, the MDPs processed by our approach perform much better in guiding the self-adaptive systems' correct operation. This further confirms the importance of meaningful updating and repairing. Currently our approach focuses on updating and repairing MDPs for self-adaptive systems, and in the future, we plan to explore its application in a wider range of domains.

References

[1] Esfahani N, Malek S. Uncertainty in self-adaptive software systems. In *Proc. the International Seminar on Software Engineering for Self-Adaptive Systems*, October 2010, pp.214-238. DOI: 10.1007/978-3-642-35813-5_9.

[2] Cámara J, Schmerl B, Moreno G A, Garlan D. MOSAICO: Offline synthesis of adaptation strategy repertoires with flexible trade-offs. *Automated Software Engineering*, 2018, 25(3): 595–626. DOI: 10.1007/s10515-018-0234-9.

[3] Weyns D, Schmerl B, Grassi V, Malek S, Mirandola R, Prehofer C, Wuttke J, Andersson J, Giese H, Göschka K M. On patterns for decentralized control in self-adaptive systems. In *Lecture Notes in Computer Science 7475*, de Lemos R, Giese H, Wüller M A *et al.* (eds.), Springer Berlin Heidelberg, 2013, pp.76-107. DOI: 10.1007/978-3-642-35813-5_4.

[4] Krupitzer C, Roth F M, VanSyckel S, Schiele G, Becker C. A survey on engineering approaches for self-adaptive systems. *Pervasive Mob. Comput.*, 2015, 17: 184-206. DOI: 10.1016/j.pmcj.2014.09.009.

[5] Wang Q. Towards a rule model for self-adaptive software. *SIGSOFT Softw. Eng. Notes*, 2005, 30(1): Article No. 8. DOI: 10.1145/1039174.1039198.

[6] Sama M, Rosenblum D S, Wang Z, Elbaum S. Model-based fault detection in context-aware adaptive applications. In *Proc. the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2008, pp.261-271. DOI: 10.1145/1453101.1453136.

[7] Filieri A, Hoffmann H, Maggio M. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proc. the 36th International Conference on Software Engineering*, May 31-June 7, 2014, pp.299-310. DOI: 10.1145/2568225.2568272.

[8] Filieri A, Hoffmann H, Maggio M. Automated multi-objective control for self-adaptive software design. In *Proc. the 10th Joint Meeting on Foundations of Software Engineering*, August 30-September 4, 2015, pp.13-24. DOI: 10.1145/2786805.2786833.

[9] Shevtsov S, Weyns D. Keep it SIMPLEX: Satisfying multiple goals with guarantees in control-based self-adaptive systems. In *Proc. the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2016, pp. 229-241. DOI: 10.1145/2950290.2950301.

[10] Cámara J, De Lemos R. Evaluation of resilience in self-adaptive systems using probabilistic model-checking. In *Proc. the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, June 2012, pp.53-62. DOI: 10.1109/SEAMS.2012.6224391.

[11] Franco J M, Correia F, Barbosa R, Zenha-Rela M, Schmerl B, Garlan D. Improving self-adaptation planning through software architecture-based stochastic modeling. *J. Syst. Softw.*, 2016, 115: 42-60. DOI: 10.1016/j.jss.2016.01.026.

[12] Filieri A, Tamburrelli G. Probabilistic verification at runtime for self-adaptive systems. In *Assurances for Self-Adaptive Systems: Principles, Models, and Techniques*, Cámara J, De Lemos R, Ghezzi C, Lopes A (eds.), Springer, 2013, pp.30-59. DOI: 10.1007/978-3-642-36249-1_2.

[13] Ghezzi C, Pinto L S, Spoletini P, Tamburrelli G. Managing non-functional uncertainty via model-driven adaptivity. In *Proc. the 2013 International Conference on Software Engineering*, May 2013, pp.33-42. DOI: 10.1109/ICSE.2013.6606549.

[14] Brechtel S, Gindele T, Dillmann R. Probabilistic MDP-behavior planning for cars. In *Proc. the 14th International IEEE Conference on Intelligent Transportation Systems*, Oct. 2011, pp.1537-1542. DOI: 10.1109/ITSC.2011.6082928.

[15] Kwiatkowska M, Parker D. Automated verification and strategy synthesis for probabilistic systems. In *Proc. the 11th International Symposium on Automated Technology for Verification and Analysis*, October 2013, pp.5-22. DOI: 10.1007/978-3-319-02444-8_2.

[16] Bartocci E, Grosu R, Katsaros P, Ramakrishnan C R, Smolka S A. Model repair for probabilistic systems. In *Proc. the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, March 26-April 3, 2011, pp.326-340. DOI: 10.1007/978-3-642-19835-9_30.

[17] Chen T, Hahn E M, Han T, Kwiatkowska M, Qu H, Zhang L. Model repair for Markov decision processes. In *Proc. the 7th International Symposium on Theoretical Aspects of Software Engineering*, July 2013, pp.85-92. DOI: 10.1109/TASE.2013.20.

[18] Kephart J O, Chess D M. The vision of autonomic computing. *Computer*, 2003, 36(1): 41–50. DOI: 10.1109/MC.2003.1160055.

[19] Sykes D, Corapi D, Magee J, Kramer J, Russo A, Inoue K. Learning revised models for planning in adaptive systems. In *Proc. the 2013 International Conference on Software Engineering*, May 2013, pp.63-71. DOI: 10.1109/ICSE.2013.6606552.

[20] Cheng B H C, Sawyer P, Bencomo N, Whittle J. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In *Proc. the 12th International Conference on Model Driven Engineering Languages and Systems*, October 2009, pp.468-483. DOI: 10.1007/978-3-642-04425-0_36.

[21] Cámara J, Garlan D, Schmerl B, Pandey A. Optimal planning for architecture-based self-adaptation via model checking of stochastic games. In *Proc. the 30th Annual ACM Symposium on Applied Computing*, April 2015, pp.428-435. DOI: 10.1145/2695664.2695680.

[22] Moreno G A, Cámara J, Garlan D, Schmerl B. Proactive self-adaptation under uncertainty: A probabilistic model checking approach. In *Proc. the 10th Joint Meeting on Foundations of Software Engineering*, August 30-September 4, 2015, pp.1-12. DOI: 10.1145/2786805.2786853.

[23] Kwiatkowska M, Norman G, Parker D. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. the 23rd International Conference on Computer Aided Verification*, July 2011, pp.585-591. DOI: 10.1007/978-3-642-22110-1_47.

[24] Clarke E M, Emerson E A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. the Workshop on Logics of Programs*, May 1981, pp.52-71. DOI: 10.1007/BFb0025774.

[25] Iftikhar M U, Ramachandran G S, Bollansée P, Weyns D, Hughes D. DeltaIoT: A self-adaptive Internet of Things exemplar. In *Proc. the 12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, May 2017, pp.76-82. DOI: 10.1109/SEAMS.2017.21.

[26] Puterman M L. Markov Decision Processes: Discrete Stochastic Dynamic Programming (1st edition). John Wiley & Sons, 1994. DOI: 10.1002/9780470316887.

[27] Sama M, Elbaum S, Raimondi F, Rosenblum D S, Wang Z. Context-aware adaptive applications: Fault patterns and their automated identification. *IEEE Transactions on Software Engineering*, 2010, 36(5): 644-661. DOI: 10.1109/TSE.2010.35.

[28] Yang W, Xu C, Liu Y, Cao C, Ma X, Lu J. Verifying self-adaptive applications suffering uncertainty. In *Proc. the 29th ACM/IEEE International Conference on Automated Software Engineering*, September 2014, pp.199-210. DOI: 10.1145/2642937.2642999.

[29] Yang W, Xu C, Pan M, Cao C, Ma X, Lu J. Efficient validation of self-adaptive applications by counterexample probability maximization. *Journal of Systems and Software*, 2018, 138: 82-99. DOI: 10.1016/j.jss.2017.12.009.

[30] Wilcoxon F. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1945, 1(6): 80-83. DOI: 10.2307/3001968.

[31] Abdi H. The bonferonni and Šidák corrections for multiple comparisons. In *Encyclopedia of Measurement and Statistics*, Salkind N (ed.), SAGE, 2007, pp.103-107.

[32] Salehie M, Tahvildari L. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 2009, 4(2): Article No. 4.

[33] Zhao T. The generation and evolution of adaptation rules in requirements driven self-adaptive systems. In *Proc. the 24th IEEE International Requirements Engineering Conference*, Sept. 2016, pp.456-461. DOI: 10.1109/RE.2016.18.

[34] Cheng S W, Huang A C, Garlan D, Schmerl B, Steenkiste P. Rainbow: Architecture-based self-adaptation with reusable infrastructure. In *Proc. the 1st International Conference on Autonomic Computing*, May 2004, pp.276-277. DOI: 10.1109/ICAC.2004.46.

[35] Chen B, Peng X, Liu Y, Song S, Zheng J, Zhao W. Architecture-based behavioral adaptation with generated alternatives and relaxed constraints. *IEEE Transactions on Services Computing*, 2019, 12(1): 73-87. DOI: 10.1109/TSC.2016.2593459.

[36] Howard R A. Dynamic Programming and Markov Processes (1st edition). The MIT Press, 1960.

[37] Sutton R S, Barto A G. Reinforcement Learning: An Introduction (2nd edition). Bradford Books, 2018.

[38] Calinescu R, Ghezzi C, Kwiatkowska M, Mirandola R. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, 2012, 55(9): 69-77. DOI: 10.1145/2330667.2330686.

[39] Su G, Chen T, Feng Y, Rosenblum D S, Thiagarajan P S. An iterative decision-making scheme for Markov decision processes and its application to self-adaptive systems. In *Proc. the 19th International Conference on Fundamental Approaches to Software Engineering*, April 2016, pp.269-286. DOI: 10.1007/978-3-662-49665-7_16.

[40] Filieri A, Grunske L, Leva A. Lightweight adaptive filtering for efficient learning and updating of probabilistic models. In *Proc. the 37th International Conference on Software Engineering*, May 2015, pp.200-211. DOI: 10.1109/ICSE.2015.41.

[41] Nahabedian L, Braberman V, D'Ippolito N, Honiden S, Kramer J, Tei K, Uchitel S. Dynamic update of discrete event controllers. *IEEE Transactions on Software Engineering*, 2018, 46(11): 1220-1240. DOI: 10.1109/TSE.2018.2876843.

[42] Ghezzi C, Greenyer J, Manna V P L. Synthesizing dynamically updating controllers from changes in scenario-based specifications. In *Proc. the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, June 2012, pp.145-154. DOI: 10.1109/SEAMS.2012.6224401.

[43] Hahn E M, Han T, Zhang L. Synthesis for PCTL in parametric Markov decision processes. In *Proc. the 3rd International Symposium on NASA Formal Methods*, April 2011, pp.146-161. DOI: 10.1007/978-3-642-20398-5_12.

[44] Cubuktepe M, Jansen N, Junges S, Katoen J P, Papusha I, Poonawala H A, Topcu U. Sequential convex programming for the efficient verification of parametric MDPs. In *Proc. the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, April 2017, pp.133-150. DOI: 10.1007/978-3-662-54580-5_8.

[45] Arming S, Bartocci E, Sokolova A. SEA-PARAM: Exploring schedulers in parametric MDPs. In *Proc. the 15th Workshop on Quantitative Aspects of Programming Languages and Systems*, April 2017, pp.25-38. DOI: 10.4204/EPTCS.250.3.

[46] Pathak S, Ábrahám E, Jansen N, Tacchella A, Katoen J P. A greedy approach for the efficient repair of stochastic models. In *Proc. the 7th International Symposium on NASA Formal Methods*, April 2015, pp.295-309. DOI: 10.1007/978-3-319-17524-9_21.

[47] Chatzieleftheriou G, Katsaros P. Abstract model repair for probabilistic systems. *Information and Computation*, 2018, 259: 142-160. DOI: 10.1016/j.ic.2018.02.019.

Wen-Hua Yang is an assistant professor in the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics, Nanjing. He received his Ph.D. degree in computer science and technology from Nanjing University, Nanjing, in 2017. His research interests include software engineering, self-adaptive software systems, and cyber-physical systems.

Min-Xue Pan is an associate professor in the Software Institute and the State Key Laboratory for Novel Software Technology at Nanjing University, Nanjing. He received his Ph.D. degree in computer science and technology from Nanjing University, Nanjing, in 2014. His research interests include software modelling and verification, software analysis and testing, cyber-physical systems, mobile computing, and intelligent software engineering.

Yu Zhou is a full professor in the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics, Nanjing. He received his Ph.D. degree in computer science and technology from Nanjing University, Nanjing, in 2009. From 2015 to 2016, he visited the SEAL (Software Evolution & Architecture Laboratory) at University of Zurich, Zurich, where he is also an adjunct researcher. His research interests mainly include software evolution analysis, mining software repositories, software architecture, and reliability analysis.

Zhi-Qiu Huang is a full professor of Nanjing University of Aeronautics and Astronautics, Nanjing. He received his B.S. and M.S. degrees in computer science from National University of Defense Technology, Changsha, in 1987 and 1990, respectively, and his Ph.D. degree in computer science from Nanjing University of Aeronautics and Astronautics, Nanjing, in 1999. His research interests include big data analysis, cloud computing, and web services.