A Proactive Self-Adaptation Approach for Software Systems based on Environment-Aware Model Predictive Control

Zhengyin Chen[1,2] and Wenpin Jiao[1,2,*]
[1]Institute of Software, School of Computer Science, Peking University, Beijing 100871, China
[2]Key Laboratory of High Confidence Software Technology (Peking University), MOE, China
chenzy512@pku.edu.cn, jwp@pku.edu.cn
*corresponding author

***Abstract*—Modern software systems need to maintain their goals in a highly dynamic environment, which requires self-adaptation. Many existing self-adaptive approaches are reactive, they execute the adaptation behavior after the goal violation. However, proactive adaptation can adapt before the goal violation to avoid adverse consequence so it has attracted more and more attention. Model predictive control is a widely used method to implement proactive adaptation. However, these works often ignore uncertainty of environment, which makes the prediction of the system inaccurate and affect the control effectiveness. Therefore, we propose an environment-aware model predictive control method. Its main idea is to add the environment state to the system model, predict the future state of the system according to the predicted environment state and the current state of the system, and solve the optimal control strategy. We use a web application simulation platform to evaluate our method. The results show that our method can achieve better adaptation results and reduce the occurrence of goal violation.**

Keywords—software adaptation; proactive adaptation; model predictive control; environment prediction

I. Introductions

Modern software systems need to complete tasks independently in a complex and dynamic environment, which makes people have high requirements for reliability, robustness, security and other trustworthy properties of software systems. In this case, self-adaptation is considered as an effective approach to improve the trustworthiness of the software systems. Self-adaptive system can continuously monitor the environment and system state, and adjust the behavior or structure of the system when the goal is violated [1]. The traditional self-adaptive system usually adopts a reactive structure, that is, the system will conduct compensation behavior after the system goal violation so that the goal can be satisfied again. In some critical tasks, reactive adaptation may miss the best adaptation opportunity, resulting in large losses. Moreover, it cannot obtain the optimal solution over a period of time so it may need frequent adaptation. Above all, it is difficult to provide a valid guarantee for the trustworthiness of the software system by reactive self-adaptation.

In recent years, proactive self-adaptation has attracted more and more researchers' attention [2][3][4][5][6]. The key difference between proactive adaptation and reactive adaptation is that reactive adaptation makes adjustment after detecting the violation of the goal, while proactive adaptation intends to avoid the possible violation in the future and make adjustment before the violation [7]. A system with proactive self-adaptive ability needs to predict the future state of the system and conduct adaptation behavior in advance to avoid adverse consequences, rather than just compensate for the adverse consequences. In many cases, for example, when the loss is unacceptable if the target is violated or the effect of adaptation has a long delay, it is more valuable to use proactive adaptation, and thus more effectively improve the trustworthiness of the software system.

Besides, control theory based methods are increasingly used in the construction of self-adaptive systems, because control theory has a good mathematical theoretical foundation and can provide formal guarantees [8]. Among control theory based methods, Model Predictive Control (MPC) is a commonly used control technique in the industry [9], in which a model is used to predict the behavior of a system over a period of time in the future. The optimal control input is obtained by solving an optimization problem with constraints to make output close to the set point. In each control loop, the above calculation process is repeated. MPC can be applied to multiple input and multiple output systems, corresponding to the situation of multiple effectors and multiple goals in software system. And the prediction of the future state of the system through the system dynamic model is consistent with the idea of proactive self-adaptation, so it is a natural idea to use MPC for proactive self-adaptation and has been implemented in some existing works [3][5][10].

However, in these MPC-based self-adaptation works, a nominal system model is generally used to model the software system [11], only considering how the current state of the system and the control signal affect the future state of the system, while ignoring the influence of other factors on the system [12]. This is usually not a problem when applied to physical devices, however, when studying software systems, due to the close interaction between the software system and the environment, the environment has a great influence on the software system. If the environment factors are not considered, the general nominal system model cannot accurately represent the behavior of the software system, which in turn affects the effectiveness of the self-adaptation. In addition, the environment itself is also dynamic and uncertain. We need to consider changes and randomness of the environment when making predictions.

Therefore, in this paper, we propose a proactive self-adaptation approach based on environment-aware model predictive control. Unlike the traditional MPC method, we explicitly predict how the environment will change in the future and use an environment-aware system dynamic model instead of a nominal system model to predict the system more accurately, which makes the control more effectively. In the design phase, we use dynamic Bayesian model to build the environment model based on the collected historical data of the environment. Moreover, we add the environment factors to the system dynamic model, and the resulting system dynamic model can predict the future state of the system according to the current system state, control signals, and environment state. At runtime, we use the model predictive control method, in each control loop, to solve a constrained optimization problem, that is, calculate a set of control inputs so that the state of the system in the future period of time is close to the set goal, in order to achieve proactive self-adaptation.

The main contributions of this paper include: Firstly, we summarize the main steps of proactive self-adaptation and propose a proactive self-adaptation approach. Secondly, we implement our proactive self-adaptation approach using environment-aware model predictive control. We take advantage of dynamic Bayesian model to model the dynamics and randomness of the environment and use the environment-aware system dynamic model to obtain more accurate system state predictions and we adopt the model predictive control method to realize the runtime control. Finally, we conduct experiments on a simulated web application to evaluate our method.

The remainder of this paper is structured as follows. In Section II we introduce background knowledge. In Section III we describe the overview of the proactive adaptation method we propose and further demonstrate the details in Section IV. We conduct experiments to evaluate our method in Section V and discuss the threat to validity in Section VI. Finally, we introduce related work in Section VII and summarize our work in Section VIII.

II. Background Knowledge

A. MAPE-K model

The conceptual model of a self-adaptive software system consists of three parts: the environment, the managing system, and the managed system [8]. The managing system includes control loops and adaptation goals. The environment refers to the external world that interacts with the system and its impact on the system can be observed. The managed system contains application software to implement the functions of the system. The managing system is the main component for the realization of the self-adaptation, it needs to monitor the state of the environment and the managed system and adjust the managed system.

A widely used classical control loop is the MAPE-K model, which represents four components, the Monitor, Analyzer, Planner, and Executor, they implement the four functions of the managing system, (i) the monitor is responsible for monitoring the environment and the system; (ii) the analyzer uses the most recent information collected to determine whether adaptation is needed; (iii) the planner makes an adaptation plan; (iv) the executor adjusts the managed system to perform the adaptation plan. At the same time, the four components share a knowledge base, which maintains information such as adaptation goals, system state, and environmental state.

B. Model Predictive Control

Model Predictive Control is a widely used modern control strategy as it offers a compromise between optimality and computation cost [13]. It uses a system dynamic model to predict the future state of a system. In each control loop, the controller requires to solve a constrained optimization problem, that is to find a set of control input signals that make the objective function optimal under certain constraints. This objective function is usually related to the state of the system in the future, which needs to be calculated through the system dynamic model. In general, MPC follows the receding horizon principle, that is, in each control loop, the controller will calculate a sequence of control signals, but the controller will only retain the first term of this sequence as a control signal and discard the rest, and repeat the calculation process in the subsequent control loop. Therefore, MPC is also known as Receding Horizon Control.

C. Dynamic Bayesian Network

Dynamic Bayesian Network (DBN) is a stochastic model that is based on probabilistic network and combines the original static network structure with time information to process time series data [14]. The advantage of using DBN for environment modeling is that it can support multiple environment variables and characterize the randomness of the environment.

Dynamic Bayesian Network is defined as follows:

First we define the static Bayesian Network (BN). There is a set of random variables $X = \{X_1, X_2, ..., X_n\}$, BN is a directed acyclic graph $BN = \{G, \theta\}$, consisting of graph structure G and related parameters θ. The nodes in the graph represent the random variables, and the edges represent the conditional dependencies between the variables. BN's parameters refer to the conditional probability table for each variable. DBN extends BN in time, using X_i^t to represent X_i at t time. DBN is usually represented as $< B_0, B_{\rightarrow} >$, as shown in Figure 1, where B_0 is a BN representing the probability distribution of random variables at the initial moment, and $B_{\rightarrow}$ representing the transfer network of two BNs at two adjacent moments. In a DBN, the conditional dependency between random variables depends not only on the variable at the current moment, but also on the value of the variable at the previous moment. The main researches in DBN are the learning problem and the reasoning problem, the learning problem is to explore the network structure and estimate the parameters in the network based on the existing data; the reasoning problem is to calculate the probability distribution of X^{t+h} with known observation value $X^1, ..., X^t$.

III. Proactive Self-Adaptation Mechanism

Classified from the temporal characteristics, self-adaptive methods can be divided into reactive and proactive [15]. And

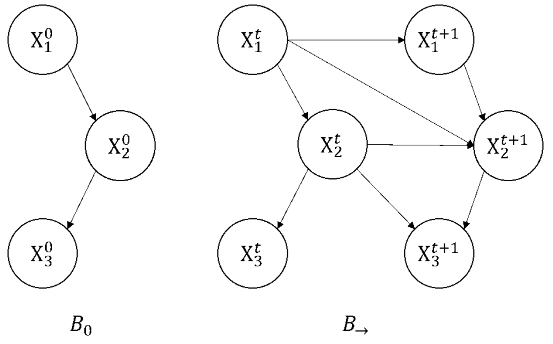


Figure 1. Dynamic Bayesian Network

most of the current self-adaptive methods are reactive, that is, the system will monitor the environment and system state, and adjust the system after the goals are violated, so that the goals are re-satisfied. This will not pose a problem when the system adapts quickly or is highly tolerant of goal violations.

However, in some scenarios, the software system needs to complete critical tasks and has a strict requirement, or the adaptation has a relatively long delay, then we hope to know the future state of the system in advance, plan and initiate adaptation early, in order to adapt more timely and reduce the loss caused by goal violations and improve the robustness, reliability and performance of the software system. The proactive self-adaptive system can predict the changes in the environment state and system state in advance, judge the completion of future system goals in advance, and make preventive actions in advance to avoid the goals violation when changes occur.

Some studies have focused on giving software systems the ability of proactive adaptation, but there is no clear definition. Combined with the existing works, we summarize the three main steps of proactive adaptation: (1) Pre-judgment, that is, the system needs to predict the state of the system over a future period of time in advance, and judge whether the system requirements can be met in the future; (2) Pre-planning, the system generates an adaptation plan in advance according to the results of pre-judgment, so that the system can avoid the occurrence of the previously predicted situation after executing the plan; (3) Pre-execution, considering the possible delay in the execution of the adaptive plan, the system often needs to initiate the plan in advance.

The overview of our proposed proactive adaptation method is shown in Figure 2. Our method follows the classic conceptual model of the self-adaptive system, adding a managing system outside the software system to ensure the continuous satisfaction of the goal. It is composed of a monitor, an adaptation decision maker which consists of an analyze-predictor and a planner, an executor and a knowledge base to complete the pre-judgment, pre-planning, and pre-execution steps of the proactive adaptation.

In the design phase, we need to learn the environment model and the system model to prepare for the proactive adaptation at runtime. For the environment model, we need to collect the historical data of the environment, and train the environment model to predict the future changes of the environment; and for the system model, it requires data sampling of the software system to obtain the output of the system under a given environment and control input to train the system model. The environment model and system model will be stored in the knowledge base for use in runtime pre-judgment and pre-planning steps.

In each control loop, the monitor needs to get the latest environment state env_0 and system state s_0 and pass them to the adaptation decision maker. The adaptation decision maker consists of an analyze-predictor and a planner, and the analyze-predictor takes env_0 and s_0 as inputs, using the system model and the environment model, to predict the value of the system state $s_{1...h}$ over the future period h, thus it completes the pre-judgment step. In the planner, we use MPC to compute the control signal. In proactive adaptation, our adaptation goal is to keep the goal of the system satisfied in the future time period h, so we define an objective function that is related to the system state in the future time period h. The planner needs to solve a constrained optimization problem and find a set of control inputs $u_{0...h}$ to make the objective function optimal, that is, the pre-planning of the adaptation behavior. According to the receding horizon principle, the adaptation decision maker transmits the first term of planning results u_0 to the executor, which adjusts the software system to complete the pre-execution of proactive adaptation.

IV. Environment-aware Model Predictive Control

A. environment model

Environment refers to the part of the external world that interacts with the system and can be measured by the system [16]. It can be divided into three categories according to its characteristics, physical environment, computing environment and user-related environment [17]. The dynamic and uncertain environment will influence the system behaviours and the satisfaction of the user goals, which makes it one of the main reasons for self-adaptation. Especially when we want software systems to have proactive adaptation ability, knowing how the environment will change and how these changes will affect the system is one of the important prerequisites for achieving proactive adaptation.

We consider the environment as a set of environment factors, and each environment factor has a constant or a variable to represent its value. In our study, we focus on the time-varying environment factor, so we use the variable ev_i^t to represent the value of environment factor i at time t. Furthermore, we define environment state as the values of all environment factors at time t: $es^t = [ev_1^t, ev_2^t, ..., ev_p^t]$, and define environment history as the sequence of environment states at time t and l time units before: $eh^t = < es^t, es^{t-1}, ..., es^{t-l} >$.

Environment prediction is the mapping from the environment history to a future environment state. Some of the existing works do not directly predict the environment, but instead observe and infer how the state of the system will change [18] or adjust the state of the system affected by the environment

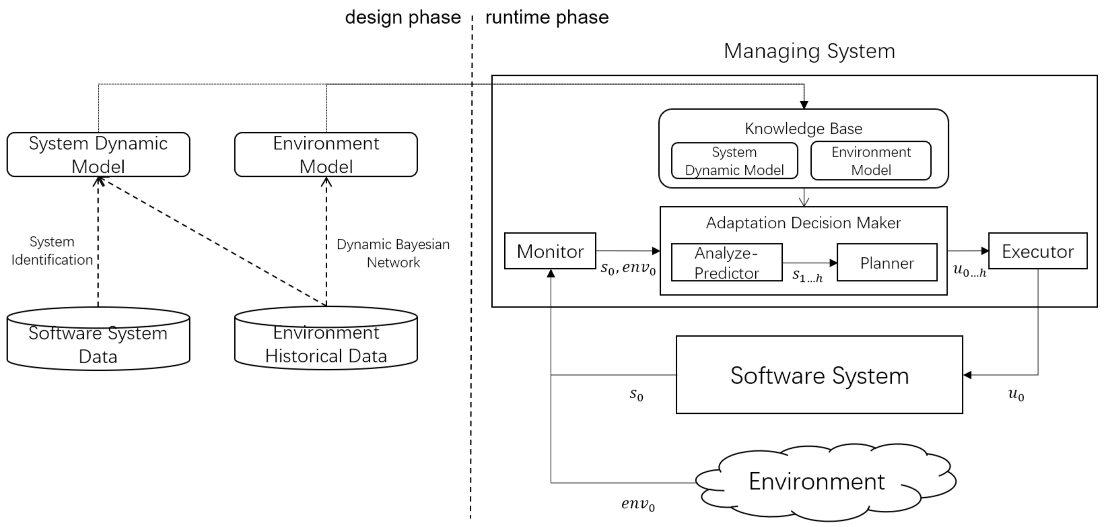


Figure 2. Overview of our proactive adaptation method

by Kalman filtering [3]. Auto-regressive methods are used for environment prediction in the work of [19]. In most cases, we cannot obtain all the knowledge about the environment, which makes it impossible to completely eliminate uncertainty, so the stochastic approach is more suitable for environment prediction.

In this paper, we use the dynamic Bayesian network to model the environment. Compared with other methods, DBN has two main advantages: (i) The environment has natural uncertainty, and DBN can describe this uncertainty in a probabilistic way. (ii) There might be multiple environment factors affecting software systems that are not independent, while DBN can make multivariate predictions.

As mentioned earlier, DBN can be expressed as $< B_0, B_{\rightarrow} >$. When using DBN to model the environment, the n environment variables we study are the random variables in DBN, that is, the X in the figure 1. For the case of $n = 1$, that is, when there is only a single environment factor, the value of the environment variable is affected only by the value of that environment variable at the previous moment, thus auto-regressive model can be used. For the case of $n \geq 1$, it is a general form of DBN, while the hidden Markov model is a commonly used model.

Environment modeling corresponds to the learning problem in DBN. It requires to collect historical data of the environment in the design phase, and to find the conditional dependence between the environment variables, as well as the dependencies between the environment variables at different times through these data, to build the network structure of the DBN and the corresponding conditional probability table. It is usually realized by applying the maximum likelihood principle. Using the learned model to predict the future state of the environment is the reasoning problem in DBN. The observation value required for the reasoning problem is the environment historical value collected by the software system at runtime, and the future probability distribution of the environment variables are obtained by the DBN inference algorithm, and the result can be obtained by unrolling the DBN by time slice. More related algorithms can be found in [14].

B. environment-aware system dynamic model

The system dynamic model represents how the system changes through time. In MPC, the system dynamic model is used to predict the future changes of the system. The system dynamic model can be divided into two categories, one is the white box analytical model by analyzing the physical behavior of the system, such as the queuing network model [20] applied in the software system, but in the software field, it is usually difficult to represent complex software behavior as an analytical model; the other one is the black-box model that is commonly used in software systems. The black-box model describes the system changes through the input-output relationship, while the internal behavior of the system is regarded as an unknown black box, and the learning method is used to fit the model.

In traditional MPC works, state space model is used for system dynamic model, which is:

$$\mathbf{x}^{t+1} = A\mathbf{x}^t + B\mathbf{u}^t \tag{1}$$

$$\mathbf{y}^t = C\mathbf{x}^t + D\mathbf{u}^t \tag{2}$$

where the superscript t indicates the time; $\mathbf{y}$ represents the system outputs, which are some measurable performance indicators in the software system; the system has n state variables, represented by the state vector $\mathbf{x} = [x_1, x_2, ..., x_n]$. The state vector $\mathbf{x}$ does not necessarily represent measurable variables, and may not even be a meaningful explanatory value for the user, but may just an abstract variable that connects input and output; $\mathbf{u}$ is a control vector $\mathbf{u} = [u_1, u_2, \ldots, u_m]$, representing the m effectors available for adjustment in the software system. A, B, C, D are coefficient matrices, in most cases, $\mathbf{y}$ and $\mathbf{u}$ are irrelevant, that is, the coefficient matrix D is a zero matrix.

The system dynamic model used in traditional MPC does not consider the influence of environment factors on the system performance. However, self-adaptive software systems usually run in a dynamic environment, and the performance of the system depends highly on the state of the environment. For example, the computational consumption required for an unmanned vehicle to plan a route in a complex terrain will be greater than in a simple terrain. If the same system model is used in the environment, the prediction of the future state of the system will inevitably be inaccurate, thus affecting the controller to generate an effective adaptation plan. Therefore, we hope to consider the environment factors in the system dynamic model.

Thus, we change the environment-aware system dynamic model to the following form:

$$\mathbf{x}^{t+1} = A\mathbf{x}^t + B\mathbf{u}^t + E\mathbf{es}^t \tag{3}$$

$$\mathbf{y}^t = C\mathbf{x}^t + D\mathbf{u}^t \tag{4}$$

$$\begin{gathered} foreach\ ev_i^t\ in\ \mathbf{es^t}, \\ V_i = \{v_1, ..., v_s\}, ev_i^t \in V_i, \\ \sum_{j=1}^{s} Pr[ev_i^t = v_j] = 1 \end{gathered} \tag{5}$$

where $\mathbf{es}$ represents the environment state, assuming we have p environment factors, then $\mathbf{es} = [ev_1, ev_2, ..., ev_p]$, E is the coefficient matrix corresponding to the environment, and the environment is added to the system state model as a separate item to reflect the relation between the system state and the environment state. And V_i represents the value range of the environment variable ev_i. Here, the uncertainty of the environment state is reflected in the discrete form of probability. It is worth noting that the continuous form of probability also works. When using this system state model for prediction, the value of the environment in the future needs to be predicted using the environment model obtained in the previous step.

This form is inspired by the system model form in Robust Model Predictive Control [11], which adds the disturbance as a separate item to the system dynamic model. The environment is similar to the disturbance, but the environment has its own characteristics. First of all, although the system cannot directly control the environment, it can obtain the state of the environment, that is to say, the environment is measurable to the system. Secondly, although the change of the environment is random, it also has certain laws, and we can predict the change of the environment. Therefore, this gives us the possibility to directly incorporate environment factors into the system dynamic model. On the other hand, the disturbance term is usually related to measurement error or noise, and needs to be considered independently for different software systems, while the environment affecting the software system is objective for all software systems.

Regarding how to build a system model, system identification [21] is a widely used method for building a black-box model. System identification is to input specific signals to the system and collect the corresponding output results, and then estimate the coefficient matrix in the system model according to the collected input and output data. When we identify the software system, first of all, we need to determine the input and output of the software system, that is, determine what $\mathbf{y}$ and $\mathbf{u}$ are in the system model, and obtain $< \mathbf{u}, \mathbf{y} >$ values as training data through simulation or other methods. The estimation process is to first determine the order of the model, that is, the number of system states x. Generally, we select the smallest order that provides the most key information through Hankel singular values. Then, it is necessary to find a set of optimal coefficient matrices to minimize the error between the system output predicted by the model and the actual system output. The possible search methods include the Gauss-Newton method, the Levenberg-Marquardt method, and the gradient descent method.

When using an environment-aware system dynamic model, the above-mentioned system identification method can also be used to estimate the system model, but two more steps are required: (1) In addition to the control input and system output, we also need to obtain the value of the environment state as an item of training data, the training data becomes in the form of $< \mathbf{u}, \mathbf{es}, \mathbf{y} >$ at this time; (2) We need to convert the model as follows, so that it can be converted into the model form required by the general system identification method. It is noted that the model of this form is only used in the stage of system identification for convenient calculation.

$$\begin{aligned} \mathbf{x}^{t+1} &= A\mathbf{x}^t + B\mathbf{u}^t + E\mathbf{es}^t \\ &= A\mathbf{x}^t + \begin{bmatrix} B & E \end{bmatrix} \begin{bmatrix} \mathbf{u}^t \\ \mathbf{es}^t \end{bmatrix} \\ &= A\mathbf{x}^t + B'\mathbf{u'}^t \end{aligned} \tag{6}$$

C. control logic

In the design phase, we obtain the system dynamic model and the environment model, and in the runtime, we can use them to build the controller to perform proactive adaptation of the system, so that the software system can keep the goal satisfaction in the dynamically changing environment.

Our controller adopts the MPC method to achieve multiple objectives by adjusting multiple control inputs. In each control loop, the controller needs to solve an online optimization problem, that is, to find a set of control inputs u^t such that the objective function is optimal, while satisfying some constraints. In MPC, we usually define the error between the predicted output and the predefined target output as the objective function to be optimized, and sometimes we need to add the control input to comprehensively consider the cost and benefit trade-offs. In general, the objective function is of the form:

$$J_t = \sum_{i=1}^{H} [\mathbf{y}^{t+i} - \mathbf{r}^{t+i}]^T Q_i [\mathbf{y}^{t+i} - \mathbf{r}^{t+i}] + \mathbf{u}^{t+i^T} P_i \mathbf{u}^{t+i} \tag{7}$$

where $\mathbf{y}^{t+i}$ is the predicted output at time $t+i$, obtained from the system dynamic model, $\mathbf{r}^{t+i}$ is the reference output, that is, the goal of the system. Q and P are the matrix used to control the weights, which can be obtained by the Analytic Hierarchy Process [22]. H is the time horizon for prediction and control.

Since randomness is added to the environment model and the system dynamic model, the design of the objective function should also consider the randomness, so the expectation form is adopted:

$$J_t = E_{es^t}\left(\sum_{i=1}^{H}[\mathbf{y}^{t+i} - \mathbf{r}^{t+i}]^T Q_i [\mathbf{y}^{t+i} - \mathbf{r}^{t+i}] + \mathbf{u}^{t+i^T} P_i \mathbf{u}^{t+i}\right) \tag{8}$$

It represents the expectation of the value of the corresponding objective function under the possible values of the environment. That is to say, the change of the environment in the future time period H is uncertain, corresponding to multiple possibilities for the state and output of the system. The objective function of this form is the expectation of the value of the objective function corresponding to each possibility.

At time t, the optimization problem that the controller needs to solve can be expressed as follows:

$$\begin{aligned}
minimize_{u^{t+i}} \quad & J_t \\
subject\ to \quad & u_{j,min} \le u_j^{t+i} \le u_{j,max}, \\
& y_{j,min} \le y_j^{t+i} \le y_{j,max}, \\
& \mathbf{x}^{t+1} = A\mathbf{x}^t + B\mathbf{u}^t + E\mathbf{es}^t, \\
& \mathbf{y}^t = C\mathbf{x}^t + D\mathbf{u}^t, \\
& foreach\ ev_i^t\ in\ \mathbf{es^t}, \\
& V_i = \{v_1, ..., v_s\}, ev_i^t \in V_i, \\
& \sum_{j=1}^{s} Pr[ev_i^t = v_j] = 1 \\
& Environment\ Model :< B_0, B_\rightarrow >
\end{aligned} \tag{9}$$

When the controller solves the solution $u^{t+i}, i = 1, ..., H$ of the above optimization problem, according to the receding horizon principle, only the first item is applied to the software system, and the subsequent items are discarded. The receding horizon principle is used because the estimation of the future system state at the current moment is not accurate, and the controller will solve the optimization problem again according to the latest obtained system state at the next moment to obtain a new solution. When in real applications, the controller may have a limited solution time, and the previously obtained solution can be saved for use when the solution cannot be obtained within the limited time.

Although the MPC uses the receding horizon principle, it can use the latest measured data for calculation in each loop, which avoids the large prediction range that results in a decrease in the prediction accuracy, and provides a certain degree of robustness. However, neither the environment model nor the system model can predict without error. Therefore, we hope to deal with the uncertainty brought by prediction using the Kalman filter. The calculation formula is as follows:

$$\hat{\mathbf{x}}^{t+1} = A\hat{\mathbf{x}}^t + B\mathbf{u}^t + K(\mathbf{y}^t - \hat{\mathbf{y}}^t) \tag{10}$$

$$\hat{\mathbf{y}}^t = C\hat{\mathbf{x}}^t \tag{11}$$

where $\hat{\mathbf{x}}, \hat{\mathbf{y}}$ are the estimated values calculated by the Kalman filter, $\mathbf{K}$ is the Kalman gain, which is used to measure the error between the predicted value and the measured value. For the calculation method of the Kalman gain, please refer to [21]. On the one hand, the Kalman filter can help us obtain the estimated value of the system state. On the other hand, since the system is affected by unknown disturbances during the actual operation, there is an error between the dynamic model of the system and the actual operation. Using the actual measured values of the system to estimate the state of the system can make the system dynamic model closer to the actual situation of the system.

V. Experiments

In this section, we will evaluate our method compared with a traditional reactive self-adaptive method and a traditional MPC method. First, we verify the SASO properties (Stability, Accuracy, Settling time, and Overshoot) in control theory under simulation conditions. Then, we use real request traces to evaluate the effectiveness of our method.

A. SWIM Case

We use SWIM [23] for evaluation, SWIM simulates a general multi-layer web application similar to Znn.com and RUBiS, such application consists of a web server layer and a database layer. The web server layer receives requests from clients. A load balancer is used to support multiple servers in the web tier. When a client requests a web page using a browser, the web server that handles the request accesses the database tier to get the data needed to render the web page. User requests arrival rate changes with time, which leads to changes in the system work load. We hope that the system can adapt to better handle this changing environment, so that the response time can be guaranteed to be within an acceptable range when the work load increases.

SWIM has two ways to deal with work load changes. First, the system can add or remove servers from the server pool connected to the load balancer. Second, the system can choose whether to return optional content (such as advertisements or recommendations), not returning optional content can reduce the computational overhead of the system. This means SWIM system can cope with the increase in load by degrading the user experience, namely, brownout-compliant [24]. SWIM uses ”dimmer” value to control the proportion of returned optional content. When the dimmer value is 1, all responses contain optional content. On the contrary, when the dimmer value is 0, all responses do not contain optional content.

In general, the system needs to keep the response time within a certain threshold. Increasing the server number and reducing the optional content can reduce the response time. At the same time, the cost of the system is related to the number of servers, and the system revenue is related to the proportion of optional content.

In order to simulate the user sending the request, we use the real network request data and scale it to fit SWIM, the system will read the timestamp of each request in the user request file, and send the request according to the arrival time interval. After reaching the load balancer, it is sent to one of the servers according to the round robin method. The server only randomly generates the service time by the normal distribution, without actually returning the content, the mean of the service time depends on whether it contains optional content and database access time. In order to simplify the calculation, in this case, resource utilization rate is used to represent the impact of database access on the service time. When resource utilization rate is high, the service time is longer. Similarly, resource utilization rate is also an environmental factor and is related to the number of user requests. After the request arrives at the server, it will be queued to wait to be processed. When the number of requests is large, the queuing time may be so long that the response time of the request may exceed the threshold. We hope that SWIM can keep the request response time within the threshold, meanwhile maximizing revenue and reducing costs as much as possible.

In this case, if we use reactive adaptation, SWIM will adjust after the response time of the request exceeds the threshold, at which time the revenue has dropped due to a large number of request timeouts. Besides, reactive adaptation only considers the current utility. When the environment is constantly changing, extra adaptations are required, and frequent adaptation behaviors will not only increase the operating cost (such as the server booting), but also affect the user experience (such as SWIM changing the proportion of optional content). Therefore, in the case of SWIM, the proactive adaptation method is used to predict changes in user requests and resource utilization rate in advance, and adjust the number of servers and dimmer value before the system violates the goal, so that the system can keep the response within the threshold for a long period. Since proactive adaptation will consider the optimal utility for a period of time, frequent adaptations will not be carried out, which increases the stability of the system operation.

B. Experiment setting

The setting for SWIM is shown in Table I. We choose the timeout rate of the request and the average response time as the outputs of the system. We set the timeout threshold to 0.75 seconds, the controllable parameters of the system are the number of servers and the dimmer value. The range of the number of servers is [1,3] and the dimmer value is in the range [0,1]. The environment factors in SWIM are the request rate and the resource utilization rate. The predict horizon is set to 3 and we get the weights Q, P using AHP.

In terms of implementation, first of all, we use python pgmpy package to implement the dynamic Bayesian network of the environment. Next, we take the environment factors into account in the system dynamic model, and use the System

TABLE I
SWIM SETTING

Name	Type	Value
Timeout rate	Output	[0,1]
Average response time	Output	-
Dimmer	Control input	[0,1]
Server number	Control input	[1,3]
Request rate	Environment	-
Resource utilization rate	Environment	[0,1]
Predict horizon H	Controller parameter	3
Weights Q	Controller parameter	[0.4 0.4]
Weights P	Controller parameter	[0.1 0.1]

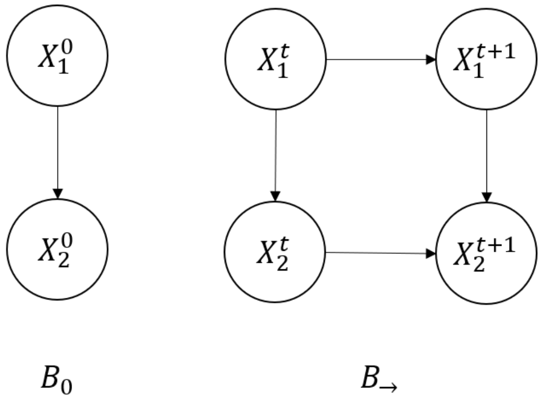


Figure 3. DBN structure of SWIM

Identification Toolbox in Matlab to obtain environment-aware system dynamic model. For the controller, we use python's do-mpc package, which uses the IPOPT package as the solver of the optimization problem.

In the first set of experiments, we use a fixed user requests number and resource utilization rate, with only one change point at runtime, to test the SASO property of our method. In this scenario, the SASO property is defined as follows, stability refers to whether the system output converges to a constant value, accuracy refers to the average response time of the system, settling time refers to the time it takes for the output to converge to a constant value, and overshoot refers to the maximum response time.

In the second set of experiments, we used ClarkNet [25] and WorldCup'98 [26] access data as request sources, which are real data collected from network nodes and websites and used in many web-related studies. We take the disk usage data characteristics of a real website as a reference to generate the corresponding resource utilization rate data for these two datasets. Since the resource utilization rate is related to the number of user requests, the DBN structure of our designed environment is as Figure 3, where X_1 represents the number of user requests, and X_2 represents the resource utilization rate.

We chose a reactive method provided by SWIM and a traditional MPC method as the baseline. The principle of the reactive method is that when the average response time exceeds the threshold, a server is added, and when it does not exceed the threshold, it increases the dimmer value. For the traditional MPC method, we choose the CobRA method [3]. CobRA uses the nominal system model to model the system without considering the impact of the environment. In the case of SWIM, it is difficult to predict the actual changes of the system based on the nominal model, so we modified the CobRA method according to the method in [27], adding environment factors, but no environment prediction.

In order to facilitate the comparison of the effect of each method, we selected three evaluation metrics, timeout rate, average response time and utility value. The timeout rate is the ratio of the number of requests that timed out to the total number of requests; the average response time is the average response time of all requests, including timeout requests; the utility value per control loop is calculated by the following formula:

$$U \triangleq U_R + U_P + U_C \tag{12}$$

where U_R, U_P, U_C represent the revenue, penalty and cost respectively, and the specific calculation is as follows.

$U_R \triangleq (1-r_t)*a*(d*R_O+(1-d)*R_M)$, where r_t represents the timeout rate, a represents the average number of requests, d is the dimmer value, R_O and R_M are the revenue brought by a single request for optional content and mandatory content, and the total revenue is the revenue that can be brought by requests that do not time out; $U_P \triangleq r_t * a * P$, where P is a negative number represents the penalty for a single timeout request. This part of the utility represents the penalty for the timeout requests; $U_C \triangleq C * (s_{max} - s)$, where C is the cost per server, s_{max} and s are the maximum number of servers and the number of servers used, this part of the utility is used to measure the cost of the server.

C. Results

The results of the first set of experiments are shown in Table II. We generated two different scenarios, one is to suddenly increase the requests number and the resource utilization rate at time 50, and the other one is the opposite, to decrease the value of these two environment factors. The initial configuration is 2 servers and 0.5 dimmer value. From the results, we can see that our method can stabilize the output of the system in two control loops. Meanwhile, it achieves good results in accuracy and overshoot properties. From the results of the SASO properties we can tell that our method has shown good control ability.

In the second set of experiments, we want to evaluate the effectiveness of our proposed environment-aware model predictive control for proactive adaptation. We compare the three adaptation methods under the two requested data traces, the results are shown in Figure 4, and other statistical data about the experiments are shown in Table III. The results of the first two control loop are not included in the statistics to eliminate the influence of the inappropriate initial setting.

TABLE II
SASO RESULTS

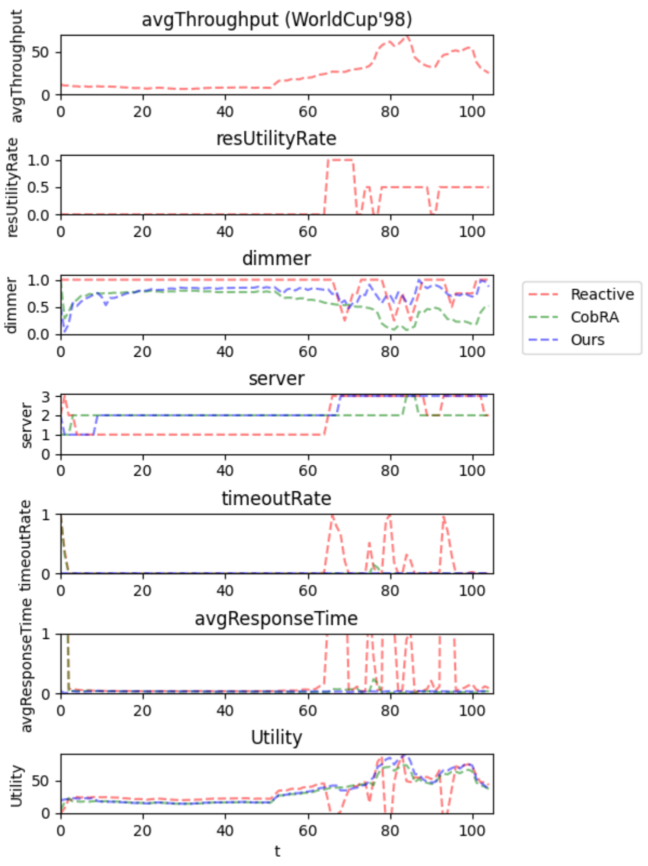
SASO property	Scenario1	Scenario2
Stability	Y	Y
Accuracy	0.025	0.015
Settling Time	2 loop	2 loop
Overshoot	0.053	0.025

We can find that in both scenarios, our method can achieve the highest cumulative utility value, and there is no obvious decrease in utility value, which indicates that our method has a good adaptation ability. Meanwhile, our method also has the lowest average timeout rate and the lowest average response time. In the ClarkNet data scenario, the pressure of user requests on the web application is relatively large, but the change range is not drastic, so the reactive method has more timeouts, while the CobRA method and our method control the number of timeouts to a low degree. In the WorldCup'98 data scenario, due to the small load pressure in the early stage of the experiment, all methods can achieve good utility values, but when the experiment progresses to about 50 time units, user requests suddenly start to rise, along with the suddenly increasing work load pressure, thanks to our method's prediction of the environment, we can choose to increase the number of servers in advance, thus avoiding the occurrence of timeouts. As for the reactive method or MPC method without environment prediction, the timeout rate increases because the server is not added in time, which reduces the utility value.

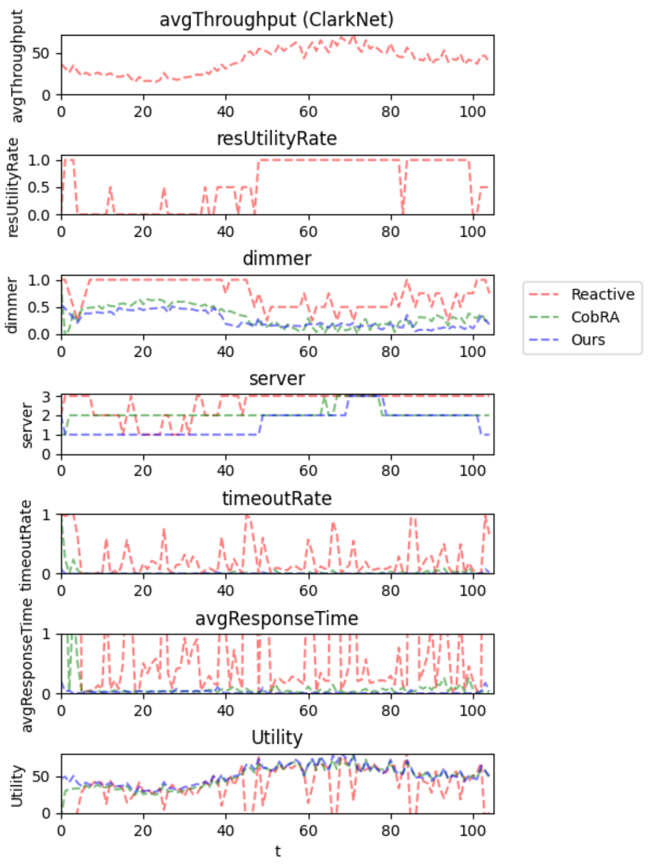
Besides, in Table III, we collect the average changing value of dimmer and server number. Although we did not consider the change of control input into the utility value, in the actual software system, frequent adaptation behavior will also lead to the increase of cost and the decrease of user experience. From the results, we can find that our method has fewer changes in the number of servers and fewer changes in dimmer value in the ClarkNet case. It enables the system to run in a relatively stable condition.

D. Summary

From the experiments, it can be found that our environment-aware model predictive control method can effectively achieve proactive self-adaptation. It has the following advantages:(1) It can avoid possible goal violations, so that the system can keep the goal satisfied, which is valuable if the violation leads to a large penalty. (2) When considering the impact of the environment on the software system and making environment predictions, the future performance of the software system can be predicted accurately and timely, which improves the effectiveness of proactive self-adaptation. (3) This method can realize adaptation with a small change of control input, so that the system can maintain stability.



(a) WorldCup'98 trace



(b) ClarkNet trace

Figure 4. SWIM results of real request traces

TABLE III
STATISTICS OF RESULTS

request trace	method	average Δdimmer	average Δserver	average timeout rate	maximum timeout rate	average response time	maximum response time	accumulated utility
WorldCup'98	Reactive	0.049	0.068	0.081	0.99	0.80	11.99	3297
	CobRA	0.028	0.029	0.00012	0.0050	0.032	0.081	3345
	Ours	0.037	0.019	0	0	0.031	0.045	3688
ClrakNet	Reactive	0.099	0.15	0.22	1.0	1.62	17.71	3891
	CobRA	0.73	0.048	0.012	0.23	0.071	1.52	5053
	Ours	0.046	0.039	0.0019	0.076	0.019	0.18	5368

VI. Threat to Validity

This section mainly discusses the internal validity and external validity of our work. A major threat to the internal validity is the parameters of MPC, such as the choice of prediction and control horizon and the choice of the objective function. A too short horizon may result in difficulty to predict the risks that the system may face, and a too long horizon will bring higher computational overhead and the risk of inaccurate prediction. While setting the objective function requires specifying the weight of the system output and control input, which is usually related to user requirements. The main threat to external validity comes from the single experimental object. We hope to select the request trace data that can represent the real situation and the changes of the data itself also have certain characteristics. At the same time, we also hope that the experiment object can represent the real software system, so the web simulator is chosen. Another aspect that affects the external validity is the scope of application of software adaptation based on control theory, which requires a model that can represent the mathematical relationship between the input and output of the software system. Because of the complexity of the software system, some software systems are difficult to express in this form or to obtain accurate mathematical relationships. Besides, our approach mainly applies to parameter-level adaptation while some software systems are more suitable for architecture-based adaptation.

VII. Related Work

In the field of software adaptation, many studies try to use control theory or control theory based methods. In [12] the authors reviewed the software self-adaptation based on control theory and proposed a classification framework from the perspective of control engineering and software engineering. Existing researches are discussed and compared from the aspects of control structure, conceptual framework, theoretical basis, development method, evaluation and verification, etc. The review summarizes the problems in the control theory based self-adaptation works, such as the difficulty in establishing an accurate model of the software system, the emphasis on control strategies but the lack of software engineering oriented methods, and the lack of consideration of uncertainty. In [28] the authors proposed the SimCA* method, which uses PID controllers to support set-point goals, optimization goals and threshold goals. It also deals with uncertainty and goal changes. The CobRA method in [3] combined MPC with requirement engineering. It expresses the goals and adjustable points of the software system through the extended requirement model, so as to control. The ProDSPL method in [5] combined MPC and feature model, and find the optimal configuration of key performance indicators through control theory while satisfying the constraints of the feature model. As mentioned above, the influence of the environment on the system is rarely considered in these works, while our work uses the prediction of the environment to obtain a more accurate system model.

In the field of proactive self-adaptation, predicting the future performance of the system is an important part. In the MPC-based method, the state-space model is generally used as the system dynamic model for prediction. It can be an analytical model [20] or a black-box model [5] obtained by learning. In our work, the learned system dynamics model is also used, and environment factors are added. The main reason for using this model is that this model can show the dynamics of the software system, that is, the influence of the previous system state on the later system state, and the learning method does not need to know the details of the system operation. The limitation of this system model is that the input and output of the system are quantifiable and measurable. Other prediction methods include: In service-based systems, the availability of a service itself and the structure of the control flow is usually used to predict the availability of the entire service [29]. The limitation of this method is the service and control flow information must be known in advance. In FUSION [30], a machine learning method is used to obtain the relationship between configuration and system performance indicators. This method shows the static relationship between configuration and system performance indicators, which is easily affected by other factors in the actual system. In the work of PLA [4], the Markov decision process is used to model the system to predict the future changes of the system, and the probability model is also used to consider the uncertainty. This method requires understanding the operating details of the software system and related modeling techniques. Some studies try to improve the effectiveness of the prediction from the perspective of improving the learning method. The study of [31] used a deep learning ensemble model, to solve the contradiction between efficiency and accuracy, while the study of [32] used a flexible analyzer to select the optimal algorithm and parameters for time series forecasting in different scenarios.

However, not all system models consider the influence of the environment. In some studies, the influence of the environment is implicitly considered, by observing changes in some behaviors of the system, inferring that how the environment changes will affect the overall operation of the system. For example, [2] use discrete-time Markov chains to model the system and its components, and then infer how the transition probabilities in the Markov chains change when the environment changes through a hidden Markov model, so as to speculate overall reliability. In [18], the authors use a label transition system to represent the behavior of the environment and the system, and then used stochastic gradient descent to update the model at runtime. In the PLA work [4], environment prediction is made explicitly, it uses the average value of the past period as the prediction of the future state, and uses the probability tree to deal with the uncertainty, but this method may not be able to achieve accurate prediction results when the environment changes drastically. PASTAA presented in [6] is based on statistical model checking, which uses historical environmental data to predict the environment, but does not model the influence of the environment on the system. Instead, they sample the environment prediction results and conduct a system simulation, evaluate the effect of the adaptation strategy and find the best strategy based on the simulation results. In our work, we directly consider the environment factors in the system dynamic model so that the impact of the environment on the system can be predicted. In terms of environment prediction, the dynamic Bayesian network can not only describe the changes of the environment in the time dimension, but can also express the relationship between multiple environment factors.

VIII. Conclusion

In this paper, we propose an environment-aware model predictive control method to realize the proactive self-adaptation of the software system. We consider the environment factors in the system dynamic model and carry out environment prediction to solve the problem in traditional model predictive control that the prediction of the system is not accurate because the environment impact is not considered. In each control loop, the controller needs to solve a constrained optimization problem and find a set of control inputs to make the system performance optimal for a period of time in the future. It is because not only the current system state, but also the expected future state of the system is considered when solving the optimization problem, which endows the software system with proactive adaptation capability.

We evaluate the effectiveness of the method through the experiments of the SWIM case. In a dynamically changing environment, using our method can effectively avoid the system's utility decline due to environment changes, so that the software system can continue to meet the requirements. Compared with the reactive method and traditional model predictive control methods without environment prediction, our method can achieve better utility value and make the system run more stably thanks to the more accurate prediction of software system performance.

The main limitation of our work is that it is difficult to obtain the system dynamic model of the software system, not all software systems can be represented in the form required in control theory. In addition, both environment prediction and system prediction are not fully accurate. Inaccurate predictions may lead to a decline in the adaptation effect or even negative effects. In future work, we will focus on how to improve the accuracy of predictions, and we hope to use our methods in more cases, and deploy it to real experimental scenarios to study its effectiveness.

ACKNOWLEDGMENT

The research was supported by National Science and Technology Major Project (2020AAA0109401), and Natural Science Foundation of China (62192731).

REFERENCES

[1] R. d. Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel *et al.*, "Software engineering for self-adaptive systems: A second research roadmap," in *Software engineering for self-adaptive systems II*. Springer, 2013, pp. 1–32.

[2] D. Cooray, E. Kouroshfar, S. Malek, and R. Roshandel, "Proactive self-adaptation for improving the reliability of mission-critical, embedded, and mobile software," *IEEE Transactions on Software Engineering*, vol. 39, no. 12, pp. 1714–1735, 2013.

[3] K. Angelopoulos, A. V. Papadopoulos, V. E. S. Souza, and J. Mylopoulos, "Model predictive control for software systems with cobra," in *2016 IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2016, pp. 35–46.

[4] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl, "Proactive self-adaptation under uncertainty: a probabilistic model checking approach," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 1–12.

[5] I. Ayala, A. V. Papadopoulos, M. Amor, and L. Fuentes, "Prodspl: Proactive self-adaptation based on dynamic software product lines," *Journal of Systems and Software*, vol. 175, p. 110909, 2021.

[6] Y.-J. Shin, E. Cho, and D.-H. Bae, "Pasta: An efficient proactive adaptation approach based on statistical model checking for self-adaptive systems," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, Cham, 2021, pp. 292–312.

[7] M. Handte, G. Schiele, V. Matjuntke, C. Becker, and P. J. Marrón, "3pc: System support for adaptive peer-to-peer pervasive computing," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 7, no. 1, pp. 1–19, 2012.

[8] D. Weyns, *An Introduction to Self-adaptive Systems: A Contemporary Software Engineering Perspective*. John Wiley & Sons, 2020.

[9] A. Filieri, M. Maggio, K. Angelopoulos, N. D'ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein *et al.*, "Control strategies for self-adaptive software systems," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 11, no. 4, pp. 1–31, 2017.

[10] L. Wang, J. Xu, H. A. Duran-Limon, and M. Zhao, "Qos-driven cloud resource management through fuzzy model predictive control," in *2015 IEEE International Conference on Autonomic Computing*. IEEE, 2015, pp. 81–90.

[11] A. Bemporad and M. Morari, "Robust model predictive control: A survey," in *Robustness in identification and control*. Springer, 1999, pp. 207–226.

[12] Q. Yang, X. Ma, J. Xing, H. Hu, P. Wang, and D. Han, "Software self-adaptation: Control theory based approach," *Chinese Journal of Computers*, vol. 39, no. 11, pp. 2189–2215, 2016.

[13] B. Kouvaritakis and M. Cannon, "Model predictive control," *Switzerland: Springer International Publishing*, vol. 38, 2016.

[14] D. Koller and N. Friedman, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[15] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM transactions on autonomous and adaptive systems (TAAS)*, vol. 4, no. 2, pp. 1–42, 2009.

[16] Y.-J. Shin, J.-Y. Bae, and D.-H. Bae, "Concepts and models of environment of self-adaptive systems: A systematic literature review," in *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2021, pp. 296–305.

[17] T. Zhao, H. Zhao, W. Zhang, and Z. Jin, "Survey of model-based self-adaptation methods," *Ruan Jian Xue Bao / Journal of Software (in Chinese)*, vol. 29, no. 1, p. 19, 2018.

[18] M. Tanabe, K. Tei, Y. Fukazawa, and S. Honiden, "Learning environment model at runtime for self-adaptive systems," in *Proceedings of the Symposium on Applied Computing*, 2017, pp. 1198–1204.

[19] J. Palmerino, Q. Yu, T. Desell, and D. Krutz, "Improving the decision-making process of self-adaptive systems by accounting for tactic volatility," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 949–961.

[20] E. Incerto, M. Tribastone, and C. Trubiani, "Software performance self-adaptation through efficient model predictive control," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 485–496.

[21] L. Ljung, *System Identification: Theory for the User*. Pearson Education, 1998.

[22] J. Karlsson and K. Ryan, "A cost-value approach for prioritizing requirements," *IEEE software*, vol. 14, no. 5, pp. 67–74, 1997.

[23] G. A. Moreno, B. Schmerl, and D. Garlan, "Swim: an exemplar for evaluation and comparison of self-adaptation approaches for web applications," in *2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2018, pp. 137–143.

[24] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez, "Brownout: Building more robust cloud applications," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 700–711.

[25] M. F. Arlitt and C. L. Williamson, "Web server workload characterization: The search for invariants," *ACM SIGMETRICS Performance Evaluation Review*, vol. 24, no. 1, pp. 126–137, 1996.

[26] M. Arlitt and T. Jin, "A workload characterization study of the 1998 world cup web site," *IEEE network*, vol. 14, no. 3, pp. 30–37, 2000.

[27] G. A. Moreno, A. V. Papadopoulos, K. Angelopoulos, J. Cámara, and B. Schmerl, "Comparing model-based predictive approaches to self-adaptation: Cobra and pla," in *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2017, pp. 42–53.

[28] S. Shevtsov, D. Weyns, and M. Maggio, "Simca* a control-theoretic approach to handle uncertainty in self-adaptive systems with guarantees," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 13, no. 4, pp. 1–34, 2019.

[29] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar, "Monitoring, prediction and prevention of sla violations in composite services," in *2010 IEEE International Conference on Web Services*. IEEE, 2010, pp. 369–376.

[30] A. Elkhodary, N. Esfahani, and S. Malek, "Fusion: a framework for engineering self-tuning self-adaptive software systems," in *Proceedings of*

the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, 2010, pp. 7–16.

[31] A. Metzger, A. Neubauer, P. Bohn, and K. Pohl, “Proactive process adaptation using deep learning ensembles,” in *International Conference on Advanced Information Systems Engineering*. Springer, 2019, pp. 547–562.

[32] C. Krupitzer, M. Pfannemüller, J. Kaddour, and C. Becker, “Satisfy: Towards a self-learning analyzer for time series forecasting in self-improving systems,” in *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 2018, pp. 182–189.