Proactive Self-Adaptation under Uncertainty: A Probabilistic Model Checking Approach

Gabriel A. Moreno
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213, USA
gmoreno@sei.cmu.edu

Javier Cámara
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
jcmoreno@cs.cmu.edu

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
garlan@cs.cmu.edu

Bradley Schmerl
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
schmerl@cs.cmu.edu

ABSTRACT

Self-adaptive systems tend to be reactive and myopic, adapting in response to changes without anticipating what the subsequent adaptation needs will be. Adapting reactively can result in inefficiencies due to the system performing a suboptimal sequence of adaptations. Furthermore, when adaptations have latency, and take some time to produce their effect, they have to be started with sufficient lead time so that they complete by the time their effect is needed. Proactive latency-aware adaptation addresses these issues by making adaptation decisions with a look-ahead horizon and taking adaptation latency into account. In this paper we present an approach for proactive latency-aware adaptation under uncertainty that uses probabilistic model checking for adaptation decisions. The key idea is to use a formal model of the adaptive system in which the adaptation decision is left underspecified through nondeterminism, and have the model checker resolve the nondeterministic choices so that the accumulated utility over the horizon is maximized. The adaptation decision is optimal over the horizon, and takes into account the inherent uncertainty of the environment predictions needed for looking ahead. Our results show that the decision based on a look-ahead horizon, and the factoring of both tactic latency and environment uncertainty, considerably improve the effectiveness of adaptation decisions.

Categories and Subject Descriptors

D.2.m [**Software Engineering**]: Miscellaneous—*self-adaptive systems*

General Terms

Design, Management, Performance

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786853

Keywords

Latency-aware, proactive, probabilistic model checking, self-adaptation

1. INTRODUCTION

Software-intensive systems are increasingly expected to operate under changing conditions, including not only varying user needs and workloads, but also fluctuating resource capacity and degraded or failed parts. Furthermore, considering the scale of systems today, the high availability demanded of them, and the fast pace at which conditions change, it is not viable to rely mainly on humans to reconfigure, and change systems as needed. Self-adaptive systems aim to address this problem by incorporating mechanisms that allow them to change their behavior and structure to adapt to changes in themselves and in their operating environment [11, 44].

Current self-adaptive systems tend to be reactive and myopic. Typically, they adapt in response to changes without anticipating what the subsequent adaptation needs will be. Furthermore, when deciding how to adapt, they focus on the immediate outcome of the adaptation. In general, this would not be a problem if adaptation tactics were instantaneous, because the system could adapt swiftly to changes, and consequently, there would not be a need for preparing for upcoming environment changes. However, many adaptation tactics are not instantaneous; that is, there is a lag between when a tactic is initiated and when the effect is produced. We call this time *tactic latency*. For example, adapting a system to shed load by producing results without including optional elements may be achieved quickly if it can be done by changing a simple setting in a component, whereas spinning up an additional server to share the load may take on the order of minutes.

Such delays are not only prevalent in modern IT systems, but are also intrinsic in other domains requiring self-adaptation. For example, some tactics used in self-adaptive wireless sensor networks require updating the firmware of the nodes [40], an operation that can take more than a minute for updating a single node [36]. Also, in a cyber-physical system, a GPS may be turned off as a power-saving tactic; however, turning it back on is not an instantaneous adaptation because the time to first fix may be about a minute [34].

In such situations, adapting *reactively* can result in inefficiencies due to the system performing a suboptimal sequence of adaptations. For example, the system may adapt to handle a transient change, only to have to adapt back to the previous configuration moments later. If the cost of performing those two adaptations is higher than their benefit, the system would be better off not adapting at all. However, reactive approaches that decide based on immediate outcomes cannot avoid such inefficiencies. This issue is exacerbated when tactics are not instantaneous. First, it may be possible that by the time the tactic completes, the situation that prompted the change has already subsided. Second, it may happen that starting an adaptation tactic prevents the system from reacting to subsequent changes until the tactic completes, rendering some adaptations infeasible for a period of time.

Ignoring tactic *latency* has negative consequences as well. Consider a situation that can be handled with one of two tactics, a or b. If tactic b is marginally better than a in terms of instantaneous utility improvement, the decision would favor tactic b. However, if tactic a is faster than b, it will start accruing the utility improvement sooner than b. Therefore, it may very well be that a is better when looking at utility accrued over time. In situations like this one, it is not possible to reason appropriately about adaptation unless the latency of adaptation tactics is considered.

Proactive latency-aware adaptation is an approach that addresses the limitations of reactive adaptation by (i) using a look-ahead horizon to proactively adapt, taking into account not only the current conditions, but how they are estimated to evolve; and (ii) explicitly considering tactic latency when deciding how to adapt, improving the outcome of adaptation [9]. Adapting proactively requires having predictions of how the environment is going to change in the near future. These predictions can come from various sources, such as time series prediction models, recurring workload patterns, or even reports from other systems that experience similar situations. Regardless of the source, these predictions will have uncertainty. Therefore, the approach must be able to decide under the uncertainty of these predictions.

In this paper, we present an approach for proactive self-adaptation under uncertainty based on probabilistic model checking, a formal verification technique used to analyze systems with stochastic behavior [30]. The approach consists of (i) creating offline formal specifications of the adaptation tactics and the system; (ii) generating periodically at run time a model to represent the stochastic behavior of the environment; and (iii) using a probabilistic model checker at run time to synthesize the optimal strategy that maximizes the expectation of a utility function over the decision horizon by analyzing the composition of the models of the tactics, the system and the environment.

The key idea is to leave the adaptation decisions in the model underspecified through nondeterminism, and have the model checker resolve the nondeterministic choices so that accumulated utility is maximized. Thanks to the use of formal specification and verification, it is straightforward for the approach to deal with the infeasibility of adaptations due to the latency of tactics, or conflicts between them. Furthermore, the same mechanism allows the adaptation decision to select multiple adaptation tactics to execute in parallel when they do not interfere with each other. We illustrate and evaluate the approach using the Rice University Bidding System (RUBiS), a web application that implements the core functionality of an auctions website [2]. To show the benefit of combining proactivity, latency awareness, and probabilistic model checking to deal with uncertainty, we compared this approach with a latency-agnostic feed-forward adaptation approach, and found that it provides considerable improvements in the effectiveness of the adaptation. Additionally, the adaptation decision time is adequate for online use, in spite of using formal verification at run time.

The rest of the paper is organized as follows. In Section 2, we introduce an example that will be used throughout the paper. An overview of the approach is given in Section 3. Section 4 provides some background on probabilistic model checking. In Section 5, we describe the core of the approach, including the formal models and how the adaptation decision is done. The evaluation of the approach is presented in Section 6. Section 8 presents related work in the areas of proactive adaptation, adaptation latency, and quantitative verification. Finally, in Section 9 we provide conclusions and future work directions.

2. EXAMPLE

To illustrate the approach we use an online auctions website as a running example. In particular, we use RUBiS [2], an open-source benchmark web application widely used for research in web application performance, and various areas of cloud computing [43, 24, 15, 19]. This multi-tier web application consists of a web tier that receives requests from clients (i.e., browsers), and a database tier. In order to support multiple servers in the web tier, we added a load balancer that distributes the requests among the web servers following a round-robin policy. The web servers access the database tier to get the data needed to render the pages with the dynamic content requested by the clients. The workload on the system depends on the request arrival rate, which fluctuates over time. This changing demand on the system constitutes its environment.

To deal with the changing load, there are two pairs of inverse adaptation tactics that can be used. The system can add or remove servers, to scale out or in, respectively. The tactic to add a server has a latency λ, which in a cloud deployment can be on the order of minutes [37]. The tactic to remove a server, however, is assumed to be immediate.[1] Additionally, the system implements the *brownout* paradigm [28]. With brownout, the response to a request includes mandatory content, such as the details of an item being browsed, and, possibly, optional content, such as recommendations of similar items. A parameter called *dimmer* is used to control the proportion of responses that include the optional content, thereby allowing the system to control the average service time for handling requests. With values in [0..1], the dimmer can be thought of as the probability of a response including the optional content. Two inverse adaptation tactics can be used to increase and decrease the dimmer value.

The cost of operating the website is proportional to the number of servers used, due to energy costs, or fees charged by cloud providers. The cost per server per unit of time is denoted by C. On the other hand, requests processed by the system provide utility or revenue. However, since companies such as Amazon, eBay, and Google claim that increased user

[1]Removing a server requires waiting for the requests already in the server to complete their processing. We assume that this time is negligible for this example.

perceived response time results in revenue loss [35], we impose a response time threshold T, such that each request whose response time is above T provides no revenue. Another factor that must be taken into account is that recommendations in e-commerce generate extra revenue [13]. Consequently, a request with the optional recommendations provides more revenue than requests without it. With R_O and R_M being the revenue of a response with the optional content and with only mandatory content, respectively, we have that $R_O > R_M > 0$.

The goal of the self-adaptive system is to maximize the difference between revenue and cost. If the system runs for a duration L, utility function (1) represents this difference:

$$U = R_O x_O + R_M x_M - C \int_0^L \big(s(t) - 1\big) dt \qquad (1)$$

where $s(t)$ is the number of servers in the system at time t, and x_O and x_M are the numbers of responses with time no larger than T, with optional and only mandatory content, respectively. Since there must be at least one server running at all times, the first server is not counted in the cost.

3. APPROACH OVERVIEW

Our approach fits in the general class of self-adaptation architectures based on explicit closed-loop control such as the monitor, analyze, plan, and execute with knowledge (MAPE-K) loop [27]. The MAPE phases cover the activities that must be performed in the control loop: (i) monitoring the system and the environment; (ii) analyzing the information collected and deciding if the system needs to adapt; (iii) planning how to adapt; and (iv) executing the adaptation. The four activities share a knowledge base or repository that integrates them. These notional elements are realized as follows in our approach:

Knowledge model. As in other architecture-based self-adaptation approaches [21], we use an abstract representation of the system that captures important system characteristics and properties as the knowledge that is used to reason about the possible adaptations. This model includes the number of servers, the number of active servers (i.e., those connected to the load balancer and able to process requests), the maximum number of servers supported, the current dimmer setting, and the observed average response time. Because some adaptation tactics have latency larger than the period of the control loop, it is necessary to keep track, in the model, of the adaptation tactics that are being executed, along with information about the progress they have made (or equivalently, when they are expected to complete). In addition, the model has information about the environment, which in this case includes the observed request arrival rate, and estimations of arrival rates in the near future.

Monitoring. Observations of the system and environment are collected, aggregated, and used to update the model. For example, the request arrival rate at the load balancer is monitored and its average and standard deviation is reflected in the model. In terms of architectural changes, when a server finishes booting and is connected to the load balancer, the monitoring marks the server as active in the model.

Adaptation Decision. Even though MAPE-K has distinct phases for analyzing the system to determine if adaptation is needed, and for planning how to adapt, these are combined into a single activity in our approach. When the goal of self-adaptation is to maximize a utility function, determining whether it is possible to adapt the system to a configuration that will give higher utility—the analysis part—implies finding such a configuration—the planning part. In our approach, the adaptation decision phase is run periodically, at a fixed interval τ. With a single invocation of the probabilistic model checker, it is possible to determine both whether adaptation is required, and what tactics should be used, if needed. The output of the adaptation decision is a (possibly empty) set of adaptation tactics to be executed.

Execution. The execution manager is a component that receives the set of tactics computed by the adaptation decision, and executes them. It executes asynchronously relative to the adaptation decision, so that if it has to execute a tactic with latency larger than the evaluation period (e.g., adding a server), the adaptation decision can still be run according to its period. Being able to do so allows the approach to complement slow tactics with fast ones if they do not interfere with each other. For example, suppose at some point, only the tactic to add a server is started because it was determined that it was going to be sufficient to handle a predicted increase in the arrival rate. However, in the following evaluation period—and before the tactic to add server completes—the realization of the environment is worse than it was estimated. In this case, the adaptation decision can instruct the execution manager to execute the tactic to decrease the dimmer value, a fast tactic. The execution manager can execute these adaptation tactics in parallel; thus, it can change the dimmer value right away, without waiting for the other tactic to complete.

The following section provides a brief background on probabilistic model checking, and Section 5 describes the core of the approach, the adaptation decision using probabilistic model checking.

4. PROBABILISTIC MODEL CHECKING

Probabilistic model checking is a set of techniques that enable the modeling and analysis of systems that exhibit stochastic behavior, allowing quantitative reasoning about probability and reward-based properties (e.g., resource usage, time, etc.). These techniques employ state-transition systems augmented with probabilities to describe the system behavior.

Moreover, probabilistic model checking approaches employing formalisms that support the specification of nondeterminism, such as Markov decision processes (MDPs), and probabilistic timed automata (PTAs), also enable the synthesis of strategies guaranteed to achieve optimal expected probabilities and rewards.

Our approach to decision-making in proactive adaptation is based on the synthesis of optimal strategies for reward-based properties in MDPs [32], since it allows us to (i) reason stochastically about uncertainty in the environment, and (ii) find optimal strategies based on a reward function that is easily mapped to maximizing utility.

Definition 1 (Markov decision process). *A Markov decision process (MDP) is a tuple $\mathcal{M} = \langle S, s_I, A, \Delta, r \rangle$, where $S \neq \emptyset$ is a finite set of states; $s_I \in S$ is an initial state; $A \neq \emptyset$ is a finite set of actions; $\Delta : S \times A \to \mathcal{D}(S)$ is a (partial) probabilistic transition function; and $r : S \to \mathbb{Q}_{\geq 0}$ is a reward structure mapping each state to a non-negative rational reward. $\mathcal{D}(X)$ denotes the set of discrete probability distributions over finite set X.*

An MDP models how the state of a system can evolve in discrete time steps. In each state $s \in S$, the set of enabled actions is denoted by $A(s)$ (we assume that $A(s) \neq \emptyset$ for all states). Moreover, the choice of which action to take in every state s is assumed to be nondeterministic. Once an action a is selected, the successor state is chosen according to probability distribution $\Delta(s, a)$.

We can reason about the behavior of MDPs using strategies (also referred to a as *policies* or *adversaries*). A strategy resolves the nondeterministic choices of an MDP, selecting which action to take in every state.

DEFINITION 2 (STRATEGY). *A strategy of an MDP $\mathcal{M}$ is a function $\sigma : S \to \mathcal{D}(A)$ s.t., for each state $s \in S$, it selects a probability distribution $\sigma(s)$ over $A(s)$.*

In this paper, we use strategies that are *memoryless* (i.e., based solely on information about the current state[2]) and *deterministic* ($\sigma(s)$ is a Kronecker function such that $\sigma(s)(a) = 1$ if action a is selected, and 0 otherwise).

Reasoning about strategies is a fundamental aspect of model checking MDPs, which enables checking for the existence of a strategy that is able to optimize an objective expressed as a quantitative property in a subset of probabilistic reward computation-tree logic (PRCTL) [3]. PRCTL extends PCTL [6] to reason about reward-based properties. A PRCTL property can state that an MDP has a strategy that can ensure that the probability of an event's occurrence or an expected reward measure meets some threshold. An extended version of the PRCTL reward operator $\mathsf{R}^{\mathsf{r}}_{\mathsf{max}=?}[\mathsf{F}^*\ \phi]$ enables the quantification of the maximum accrued reward r along paths that lead to states satisfying the state formula ϕ. A typical example of a property employing the reward maximization operator is $\mathsf{R}^{\mathsf{time}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{empty\_battery}]$, meaning "maximum amount of time that a cell phone can operate before its battery is fully discharged."

In the following section, we show how similar properties that refer to utility-based rewards are employed for decision-making in the context of proactive adaptation.

5. ADAPTATION DECISION

At a high level, the adaptation decision answers the question of what adaptation tactic(s) should be started now, if any, to maximize the aggregate utility that the system will provide in the rest of its execution. Poladian et al. showed that reacting to the current situation without looking ahead can result in suboptimal solutions when there is an adaptation cost [42]. We argue a similar case for situations where adaptations have latency, even if there is no adaptation cost. When there is no adaptation cost, tactics do not directly affect utility. Rather, they change the system configuration, which in turn results in a change in utility. If adaptation tactics had no latency, the system could adopt any configuration anytime, and thus no look-ahead would be necessary for optimal adaptation decisions. However, when tactics have latency, it takes some time for the system to adapt to a new configuration. Therefore, the configuration of the system at time t constrains the possible configurations at a later time $t + \tau$, if τ is smaller than the latency of at least one of the adaptation tactics. For instance, if the current configuration has one server at time t, and $\tau < \lambda$, then all system configurations with more than one active server are not feasible at time $t + \tau$. Consequently, it is not possible to find the best configuration, or the adaptation to get to it, without looking ahead to see which configurations will be needed in the future.

Although the decision approach must look ahead, it is not practical to look too far into the future because of computational complexity, and because the uncertainty of the environment predictions increase as they get further into the future. Therefore, the adaptation decision uses look-ahead with a finite horizon, and the question it answers is what adaptation tactics should be started now, if any, to maximize the aggregate utility the system will provide over the horizon.

The overall approach to solve the adaptation decision problem using probabilistic model checking is to analyze the MDP model that results from the parallel composition of processes representing the behavior of the environment and the system, starting at the current time until the horizon. These models are abstractions that contain only the properties of the system and the environment that are necessary to compute the value of the utility function, and to keep track of how the system changes when tactics are applied. The key idea is to leave the decision to execute adaptation tactics underspecified in the model through nondeterministic behavior. The probabilistic model checker PRISM [31] is then used to synthesize a strategy that maximizes the expected accumulated utility over the horizon. Since the strategy is a resolution of the nondeterminism in the model, it indicates which tactics must be used and when.

The following sections elaborate on the overall structure of the model used to decide; describe the models of the environment, system, and tactics; and provide more details about how the model checker is used with these models to solve the adaptation decision problem.

5.1 Overall Model Structure

The model used for the decision describes the behavior of the adaptive system in the context of the predicted behavior of the environment over the look-ahead horizon. As depicted in Figure 1, it is composed of modules (or equivalently, concurrent processes) for the environment, the adaptation tactics, and the system. The orchestration of these processes is critical to get the right behavior. However, it is as important to leave enough nondeterminism in the scheduling of the processes to give the model the freedom to decide when to use the adaptation tactics. The orchestration is accomplished via a module, clk, that controls the passing of time, and through the synchronization of modules on shared actions (the connectors in the figure).

The overall schedule of how processes should be scheduled is as follows. The execution of the model is done at the granularity of evaluation periods, so one unit of model time corresponds to τ in real-world time. Time 0 in the model represents the beginning of the look-ahead horizon (i.e., the current time in the controlled system). At the beginning of each evaluation period in the execution of the model, the system must have a chance to proactively adapt. Once the system has adapted (or just passed the opportunity), the environment updates its state for the current period by taking a probabilistic transition according to its model. After that, the utility that the system provides for the period is computed, and accumulated. Then, time is advanced, and the process is repeated until the end of the horizon is reached.

[2]However, time is explicitly encoded as part of the state.

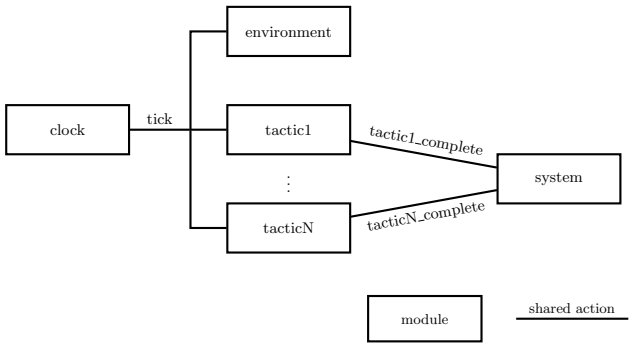

Figure 1: Module composition in adaptation decision model

```
1  module clk
2      time : [0..HORIZON + 1] init 0;
3      readyToTick : bool init true;
4
5      [tick] readyToTick & time < HORIZON + 1 -> 1 : (time' = time
           + 1) & (readyToTick'=false);
6      [tack] !readyToTick -> 1 : (readyToTick'=true);
7  endmodule
8
9  rewards "util"
10     [tack] true : UTILITY_SHIFT + periodUtility;
11 endrewards
```

Listing 1: Clock module and reward structure

The specification of the clk module is shown in Listing 1. At each period, it takes two transitions. First the command labeled with the action tick advances the time. However, the environment and the tactics share the same action, and since a module can only execute a labeled command when all modules sharing the same label execute their corresponding command synchronously, clk will only be able to advance the time when the tactics and environment modules are ready to do so. After that happens, clk takes another transition labeled tack, with which the utility accumulation synchronizes.[3] This ensures that neither the system nor the environment change when utility calculation is done. The reward structure util (lines 9-11) defines the reward function that will be maximized by the model checker, which in this case is the accumulation of the utility for the periods in the decision horizon.[4] Although not shown in the listing, periodUtility is a formula that encodes (1) for a single period.

5.2 Environment Model

The goal of the adaptation decision is to decide how to adapt to maximize the utility the system will accrue over the look-ahead horizon. However, utility is a function of both the system configuration and the environment state. Therefore, deciding with a look-ahead horizon requires predicting the near future states of the environment. These predictions are not perfect, though, and, consequently, they are subject to uncertainty. In Esfahani and Malek's list of sources of uncertainty that affect self-adaptive systems, this corresponds to uncertainty of *parameters over time* [16].

The environment can be modeled as a stochastic process in which the random variable representing the state of the environment has one realization at each time step, with a time step being equal to the evaluation period τ. More specifically, we want to model the evolution of the environment over the decision horizon as an MDP, so that it can be composed with the model of the system and tactics for the analysis. Note that the specification of the environment model does not include any nondeterminism, making the description of the environment's behavior fully probabilistic (i.e., analogous to a discrete-time Markov chain[5]). As will be explained later, the resolution of the nondeterminism in the adaptation tactics is used to decide how the system should adapt given the probabilistic behavior of the environment.

In our example, the environment predictions are made using an autoregressive (AR) time series predictor that is part of the RPS toolkit [14]. The monitoring component measures the interarrival time between requests arriving at the load balancer. At the beginning of each evaluation period, the knowledge model is updated with the average interarrival time for the previous period. This observation is supplied to the time series predictor so that it can update its internal model. Using the predictor, it is possible to obtain estimations for the average interarrival time for the next evaluation period, given the past observations. Since the estimation has an error with a normal distribution, the predictor provides the variance associated with the estimation.

To create an MDP that captures both the prediction of the environment states and its uncertainty, we construct a probability tree. The root of the tree corresponds to the current state of the environment, each node represents a possible realization of the environment, and its children represent realizations conditioned on the parent, with the edges representing the probability of the child realization given that the parent was realized. Creating a small number of branches at each node requires discretizing the probability distribution of the estimation for the following period. Usually, three-point discrete-distribution approximations are used for constructing probability trees for decision making. In our approach, we use the Extended Pearson-Tukey (EP-T) three-point approximation [26]. This approximation consists of three points that correspond to the 5th, 50th, and 95th percentiles of the estimation distribution, with probabilities 0.185, 0.630, and 0.185, respectively.

The construction of the probability tree starts with the root, which is the current state of the environment, e_0 in Figure 2. Using the predictor, we obtain the distribution for the estimation for the following period, $\hat{e}_1$. Note that the predictor has already seen the past realizations of the environment, up to e_0, so the prediction is implicitly conditioned on the past observations. Using the estimation $\hat{e}_1$, and the EP-T discrete approximation, three children of the root are created. In Figure 2, the nodes $P_k(\hat{e}_1)$ represent the kth percentile of the distribution of the estimation $\hat{e}_1$. To continue the expansion of the tree, each child is visited and its children created in the same way. However, the estimation for these children must be conditioned on the parent. This

[3]This means that the utility for the period is computed right after time is advanced, which is different than the conceptual schedule described before. This change allows a reduction of the state space without affecting the resulting decision.

[4]The utility function is translated with a large positive shift constant because PRISM does not allow negative rewards. This does not affect the result of the optimization.

[5]Specifically, this process is an MDP in which only one action can be selected in each state, the outcome of which is governed by a single discrete probability distribution.

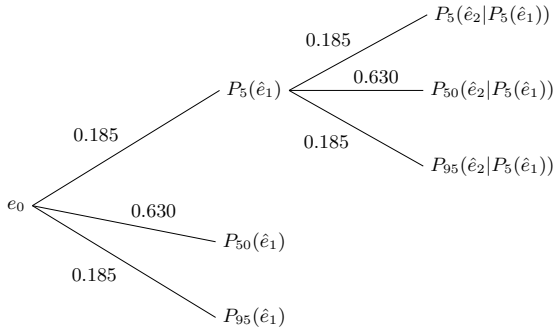

Figure 2: Probability tree

is achieved by cloning the predictor (to avoid disturbing the state of the original predictor), and giving to it the state of the environment at the parent, as if that state would have actually been the realization of the environment. In that way, when we obtain the prediction for the following period, the prediction will be conditioned on the parent.

In principle, it would be possible to continue the expansion of the probability tree up to a depth equal to the length of the look-ahead horizon. However, the further into the future we get (or the deeper in the tree), the higher the uncertainty of the predictions, and the larger the resulting state space. In their use of probability trees, Poladian et al. found that they could limit the branching depth without much impact on the quality of the solution [42]. We take a similar approach, limiting the branching in the tree to two levels, and, beyond that, continuing the extension of the branches without any further branching up to a depth equal to the horizon.

A new probability tree is generated at the beginning of the adaptation decision phase of each period. Generating the PRISM code that represents the MDP for the probability tree is straightforward. Listing 2 shows its specification in PRISM. Each node of the probability tree is assigned a unique number in the range $[0..N-1]$, where N is the number of nodes. A variable s in the env module with the same range represents the state of the environment in the probability tree. The transitions out of each node can be encoded directly as commands in PRISM.[6]

The action tick is used to synchronize the transitions of the environment with the transitions of the clock and the system. The mapping from state s to the value of the state is encoded in the formula starting in line 9, using the conditional operator. In this formula, constants such as P5_E_1 represent the values of the nodes of the probability tree.

5.3 System and Tactics Models

In addition to the environment state, the system configuration is also needed to compute the value of the utility function the adaptation decision is maximizing. Unlike the environment model, the model of the system and its tactics

[6]MDPs are encoded in PRISM with commands like:
$[action]$ *guard* -> $p_1 : u_1 + \ldots + p_n : u_n$
where *guard* is a predicate over the model variables. Each update u_i describes a transition that the process can make (by executing *action*) if the guard is true. An update is specified by giving the new values of the variables, and has an assigned probability $p_i \in [0, 1]$. Multiple commands with overlapping guards (and probably, including a single update of unspecified probability) introduce local nondeterminism.

```
1  module env
2  s : [0..N−1] init 0;
3
4  [tick] s = 0 −> 0.185 : (s' = 1) + 0.63 : (s' = 2) + 0.185 : (s' = 3);
5  [tick] s = 1 −> 0.185 : (s' = 4) + 0.63 : (s' = 5) + 0.185 : (s' = 6);
6  ...
7  endmodule
8
9  formula stateValue = (s = 0 ? E_0 : 0) +
10                      (s = 1 ? P5_E_1 : 0) +
11                      (s = 2 ? P50_E_1 : 0) +
12                      ...
```

Listing 2: Environment module in PRISM

```
1  module sys
2      servers : [1..MAX_SERVERS] init INI_SERVERS;
3      dimmer : [1..DIMMER_LEVELS] init INI_DIMMER;
4
5      [addServer_complete] servers < MAX_SERVERS −> 1 : (servers'
           = servers + 1);
6      [removeServer_complete] servers > 1 −> 1 : (servers' = servers −
           1);
7      [increaseDimmer_complete] dimmer < DIMMER_LEVELS −> 1 :
           (dimmer' = dimmer + 1);
8      [decreaseDimmer_complete] dimmer > 1 −> 1 : (dimmer' =
           dimmer − 1);
9  endmodule
```

Listing 3: System module

does not change at run time except for a few initialization constants; and consequently, it can be constructed offline.

The system model only has to keep track of the configuration information that is needed as input to the utility function. In the case of our example system, this information includes the number of active servers, and the value of the dimmer. This model does not actually model the processing of individual requests. Instead, it uses a queueing theory model to compute the average response time for each period based on the system configuration and the environment state. Because web servers can only handle a limited number of requests simultaneously, we use a limited processor sharing (LPS) model [47], which considers a system in which the number of concurrent requests that can be processed by each server simultaneously is limited by a constant K. We set K equal to the maximum number of processes configured for each server in the system.

The system model must also reflect the changes in its state that would result from the use of adaptation tactics. For better modularity, each adaptation tactic is modeled as a separate PRISM module that synchronizes with the system module on actions that represent the completion of the adaptation tactic. The specification of the system module is shown in Listing 3. Lines 2-3 have the two variables that capture the system configuration. They are initialized with constants INI_* that represent the state of the system at the time the adaptation decision is invoked; that is, at the beginning of the decision horizon. Lines 5-8 have commands that capture how the system state is updated when each of the adaptation tactics completes. Since each command is synchronized with the completion of the corresponding tactic, the associated state updates can only take place when the tactic completes.

The responsibilities of each tactic module include determining if the tactic's applicability conditions are met, deciding

nondeterministically whether to start the tactic or not, keeping track of the progress of the tactic (if it has latency), and synchronizing with the system module when the tactic completes. Listing 4 shows the model for the tactic to add a server. Line 1 computes the number of evaluation periods that correspond to the latency of the tactic, since the tracking of the latency of tactics is done at the granularity of periods. The predicate in line 4 expresses the conditions under which the tactic applies, that is, that we have not yet used all available servers and that the tactic can be executed concurrently with other tactics currently executing (expressed here in addServer_compatible). This latter condition allows the adaptation decision to decide to execute non-conflicting tactics concurrently (e.g., decreasing the dimmer value while a server is being added), while avoiding the concurrent execution of conflicting tactics (e.g., removing a server while one is being added). The state of the tactic is defined in lines 7-8. The variable addServer_state is used to keep track of whether the tactic is executing or not (it is greater than 0 when the tactic is executing), and if it is, how much progress it has made. As was the case with the system state, the state of this variable is initialized with a constant that represents its state at the time the adaptation decision is invoked. This is needed because the tactic may already be in progress when the adaptation decision is carried out, and that must be taken into account to avoid making decisions inconsistent with the state of the system. This is the reason why the knowledge model needs to keep track of tactic execution.

The variable addServer_go is for internal bookkeeping, and has to do with the orchestration of the modules. At the beginning of each period, this variable is true, as it is the sys_go predicate, which is simply an alias to readyToTick from the clock module. This means that the tactic has an opportunity to execute one of its enabled behaviors. After doing that, addServer_go is set to false to force the tactic to wait until the next period by leaving only the transition labeled with tick enabled (line 38). When the tactic is enabled and applicable, the two commands starting in lines 11 and 17 are enabled with identical guards. These commands correspond to starting the execution of the tactic (line 14), and just passing (line 20). Since they have no probability specified on their right-hand side, the model has a nondeterministic choice between them. When the tactic is enabled, but it is not applicable, it passes (lines 23-26). The commands starting in lines 29 and 34 model the progress of the tactic, and its completion, respectively. The latter must synchronize with the system module on the action addServer_complete, causing the system to reflect the change caused by the completion of the tactic in its configuration. Note that after the completion of the tactic, addServer_go is left true to make it possible for the tactic to start again in the same period.

The rest of the tactics are defined in a similar way, and since they are modeled as concurrent processes that synchronize only when they all have had a chance to execute, their ordering is nondeterministic. This, combined with the nondeterminism in the decision to start each tactic, give sufficient flexibility to the model checker so that it can decide how to best schedule the adaptation tactics.

```
1  const int addServer_LATENCY_PERIODS = ceil(addServer_LATENCY
       / PERIOD);
2
3  // applicability conditions
4  formula addServer_applicable = servers < MAX_SERVERS &
       addServer_compatible;
5
6  module addServer
7      addServer_state : [0..addServer_LATENCY_PERIODS] init
           ini_addServer_state;
8      addServer_go : bool init true;
9
10     // tactic applicable, start it
11     [addServer_start] sys_go & addServer_go // can go
12         & addServer_state = 0 // tactic has not been started
13         & addServer_applicable
14         -> (addServer_state' = 1) & (addServer_go' = false);
15
16     // tactic applicable, but don't start it
17     [] sys_go & addServer_go // can go
18         & addServer_state = 0 // tactic has not been started
19         & addServer_applicable
20         -> (addServer_go' = false);
21
22     // pass if the tactic is not applicable
23     [] sys_go & addServer_go
24         & addServer_state = 0 // tactic has not been started
25         & !addServer_applicable
26         -> 1 : (addServer_go' = false);
27
28     // progress of the tactic
29     [] sys_go & addServer_go
30         & addServer_state > 0 & addServer_state <
               addServer_LATENCY_PERIODS
31         -> 1 : (addServer_state' = addServer_state + 1) &
               (addServer_go' = false);
32
33     // completion of the tactic
34     [addServer_complete] sys_go & addServer_go
35         & addServer_state = addServer_LATENCY_PERIODS //
               completed
36         -> 1 : (addServer_state' = 0) & (addServer_go' = true); //
               so that it can start again at this time if needed
37
38     [tick] !addServer_go -> 1 : (addServer_go' = true);
39 endmodule
```

Listing 4: Tactic to add a server

5.4 Adaptation Decision

The adaptation decision can be carried out after the environment model has been constructed as described in Section 5.2. The input to the probabilistic model checker is the composition of the modules previously described. Because we want the model checker to synthesize a strategy, we also have to specify the property of the model that must hold under the generated strategy. In this case, the desired property is to maximize the accumulated utility over the look-ahead horizon. In PRCTL, this property is expressed as

$$R^{\mathsf{util}}_{\mathsf{max}=?}[F^{c}\mathsf{end}]$$

where util is the reward structure specified in the model (Listing 1, lines 9-11), and end is a predicate that indicates the end of the look-ahead horizon.

The strategy synthesized by PRISM resolves the nondeterminism in the model, replacing nondeterministic choices with choices based on the state of the system and the environment. Because the behavior of the environment remains stochastic, it is not possible to extract from the strategy what adaptation tactics should be used at each time step in the horizon, since that decision depends on the stochastic behavior of the environment. That notwithstanding, the choices made by the strategy at time 0 are deterministic because they are made before the environment takes any probabilistic transition. Because these choices are exactly the ones that should be enacted at the current time in the controlled system (recall

that time 0 in the model corresponds to the current time), it is sufficient to extract these from the strategy and ignore future choices. The set of tactics extracted from the synthesized strategy are handed off to the execution manager, thus completing the adaptation decision phase.

6. EVALUATION

The approach was evaluated using a version of RUBiS extended by Klein et al. to support brownout [28].[7] The evaluation setup consisted of two computers: an Intel Core i7-4800MQ quad-core processor at 2.7GHz and 16GB of RAM, where the self-adaptive system was deployed, and another computer to generate traffic to the website. The website setup had up to three web servers, each running in a virtual machine (VM) hosted in the main computer. Each VM had 4GB of RAM, and one virtual CPU pinned to a dedicated core. The host OS, Ubuntu 14.04 LTS, was configured to isolate (and thus not use) the cores dedicated to the VMs. The load balancer, HAProxy [1], was run in the host OS, and configured to distribute requests with a round-robin policy among the available web servers. The adaptation layer (monitoring, adaptation decision, execution manager, and knowledge model) was also run in the host OS. In order to keep the latency of adding a server controlled in the experiments, the three VMs were kept running during each experiment run, and the booting and shutting down of a VM was simulated by enabling and disabling, respectively, the VM in the load balancer. The latency of the tactic to add a server was simulated by the execution manager by imposing a delay λ between the start of the tactic, and the time the server was enabled and marked active.

The parameters of the utility function (1) were assigned as follows. The server cost was assumed to be $C = 1$ monetary unit per second, and the rest of the parameters were derived from it. The average time to serve a request with and without the optional content was measured profiling the website with a client making a single request at a time so that there was no queueing time involved. Using the queueing model, it was determined that with a response time threshold $T = 1$ second, a single server was able to handle 53.8 requests per second with the optional content. Assuming that the cost of a server can be covered by the revenue of handling half its maximum capacity with optional content, the revenue for a response with optional content was computed as $R_O = \frac{2}{53.8}C$. The maximum capacity for handling requests without the optional content was 921.8 requests per second, but in this case we require that 2/3 of the server capacity be used to cover its cost; that is, $R_M = \frac{3/2}{921.8}C$. In that way, the utility function reflects the notion that it should be preferred to use server capacity for providing optional content, and not just stay with the minimum number of servers and deal with load changes using the dimmer.

The evaluation period τ was configured to be 60 seconds. The length of the look-ahead horizon was determined as $h = \max\left(5, \left\lceil \frac{\lambda}{\tau} \right\rceil (S_{max} - 1) + 1\right)$, where $S_{max} = 3$ is the maximum number of servers. This computes a horizon that is long enough for the system to go from one server to the maximum number of servers plus one period to observe the benefit. The minimum of 5 is used to enforce look-ahead even if the tactic latency is small.

[7] `https://github.com/cristiklein/brownout`

The adaptation tactics were those described in Section 2, with the number of servers allowed to range between 1 and 3, and the dimmer levels allowed to take values 0, 0.25, 0.5, 0.75, and 1. To use both conflicting and non-conflicting tactics, each pair of inverse tactics was specified as conflicting with itself, and non-conflicting with the other. That means that, for example, a server could not be removed while one was being added, but the dimmer value could be increased or decreased while the number of servers was being changed.

To get realistic arrival patterns, we used an arrival trace from requests made to the World Cup '98 website [4] during one day of the tournament. This trace has considerable load increases around the two games that were played on that day. We took the arrivals between noon and midnight (thus encompassing the two games), and scaled it to last 75 minutes, and not to exceed the maximum capacity of the evaluation setup. To replay the trace, we used a single client that was able to send as many concurrent requests as needed to reproduce the traffic of the trace. Only the timestamps of the trace were used, because all the requests in the experiments targeted the same URL, which randomly selected an item to render its details page.

Besides the proactive latency-aware (PLA) approach presented in this paper, a second approach was implemented for comparison. The latter used feed-forward (FF) adaptation in the following manner. Each evaluation period, the adaptation decision gets the estimated arrival rate for the following period, and, using the queueing model, finds the adaptation that maximizes utility for the following period, by considering all the adaptations that are possible at that time. Unlike PLA, FF is latency-agnostic. Both approaches use the same queuing model, and, although limited to one period in the future, the environment prediction used by FF is the same as the one used for PLA. To check the implementation, we forced PLA to have a horizon of one period with no branching, and both produced exactly the same results.

For each adaptation approach, the simulation was run once for each latency of the tactic to add a server, with the latter, λ, having values 60, 120, 180, and 240 seconds. In each run, the first 15 minutes were used to prime the environment time series predictor, and the metrics to do the evaluation were collected in the remaining hour. Figure 3 summarizes the results obtained with the adaptation approaches PLA and FF. Although utility is the main metric for comparison, the plots also show the average number of servers used, the percentage of responses that included the optional content, and the percentage of responses that were late. The results show that PLA performs better with respect to the four metrics. Even though both approaches show a decline in utility as the tactic latency increases, the drop shown by FF is more pronounced. In addition, PLA managed to keep the percentage of late responses very low, whereas in FF the number of late responses increased with the tactic latency.

Although the running time of the adaptation decision may be a concern due to the use of probabilistic model checking, we found that except for a few outliers, it was under 3 seconds. The box plot in Figure 4 shows the median, 1st and 3rd quartile in the box, and the range in the bar. The first decision in each run took close to 5 seconds, probably because the model checker was not cached yet; thus, the first data point for each run was not used for the plot. Taking into account that the evaluation period was 60 seconds, these decision times are acceptable. The running time of the model

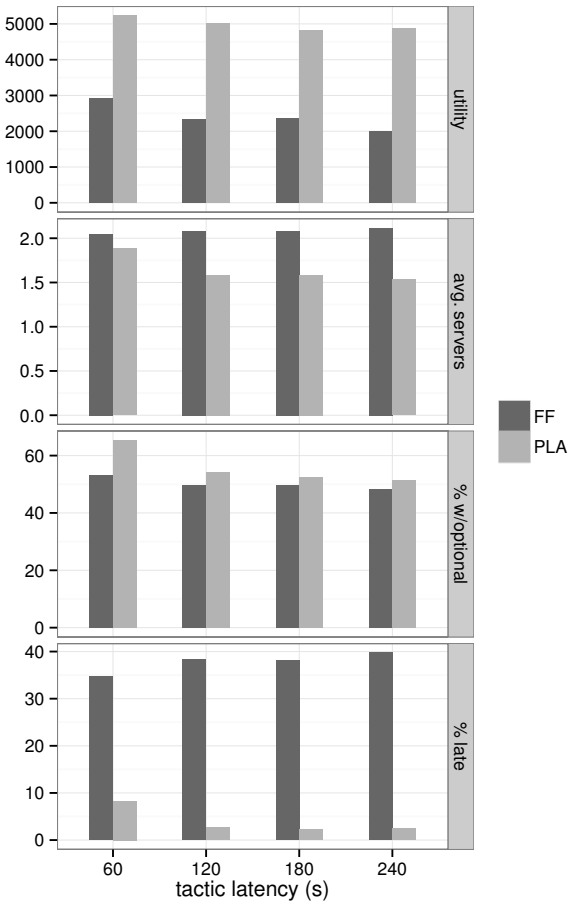

Figure 3: Evaluation results

checker increases with the number of states in the model, which, in turn, increases with the number of tactics, the latency of the tactics, the length of the horizon, and the branching and depth of the probability tree.[8] A study of the sensitivity of the decision time to these parameters is left for future work. However, it is worth noting that no optimization was done for this work. Techniques such as those proposed by Gerasimou et al. [22] could be used to reduce the adaptation decision time.

7. THREATS TO VALIDITY

The main threat to validity is the number of parameters involved that could affect the results, including the period λ, the length of the horizon, and most notably, the utility function. Although it is possible to construct a utility function that would result in no advantage for a proactive latency-aware approach, we have described the rationale for the utility function used, which we believe to be reasonable. Another threat to validity is the use of a single target system for the evaluation.

[8]Note that the horizon is the same when the tactic latency is 60 and 120 in these experiments. That explains why there is not much difference between their decision time in Figure 4.

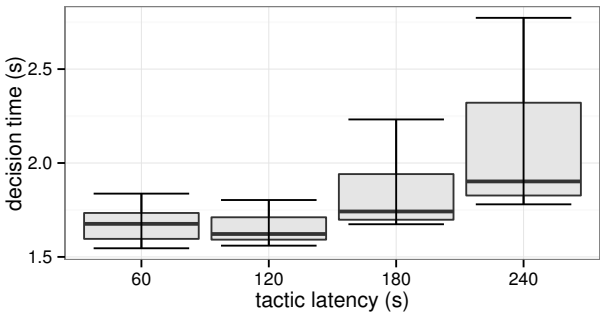

Figure 4: Adaptation decision running time

8. RELATED WORK

8.1 Proactive Adaptation

One defining characteristics of autonomic or self-adaptive systems is being *anticipatory*, defined as "[being] able to anticipate to the extent possible, its needs and behaviors and those of its context, and [being] able to manage itself proactively" [41]. Notwithstanding, the vast majority of the self-adaptive approaches are reactive [44, 29], and in their recent survey, Krupitzer et al. highlight proactive adaptation as a research challenge in the area of self-adaptive systems [29].

One area in which proactive adaptation has received considerable attention is service-based systems [8, 23, 33, 39, 45] because of their reliance on third-party services whose quality of service (QoS) can change over time. In that setting, when a service failure or a QoS degradation is detected, a penalty has already been incurred, for example, due to service-level agreement (SLA) violations. Thus, proactive adaptation is needed to avoid such problems. Hielscher et al. proposed a framework for proactive self-adaptation that uses online testing to detect problems before they happen in real transactions, and to trigger adaptation when tests fail [23]. Wang and Pazat use online prediction of QoS degradations to trigger preventive adaptations before SLAs are violated [45]. Some limitations of these approaches are that they ignore the adaptation latency, and that their look-ahead is limited, for example by considering only the predicted QoS of services yet to be invoked in a service composition being executed.

The work on anticipatory dynamic configuration by Poladian et al. [42] is the closest to ours. They demonstrated that when there is an adaptation cost, anticipatory adaptation outperforms reactive adaptation. By leveraging environment predictions and using a look-ahead horizon, anticipatory adaptation can determine the best adaptation to carry out in order to maximize the utility accumulated over time, taking into account how the environment state will evolve in the short term, and the penalties associated with adapting. One limitation of this work is that it ignores adaptation latency, which has the following consequences: (i) it cannot select between fast and slow adaptations, (ii) it cannot be proactive because it cannot start adaptations with the necessary lead time to complete by the time the environment changes, and (iii) it assumes that all configurations are feasible at all times.

8.2 Adaptation Latency

Adaptation latency (i.e., how long the system takes to adapt) is a concern in autonomic computing, and has been

proposed and used as a metric to evaluate adaptation approaches [10, 5, 18, 38]. Nevertheless, it is rarely taken into account as a factor in the adaptation decision. As Gambi et al. point out, adaptations are typically assumed to be immediate. So, they pose—but not address—the research question of how knowledge of adaptation latency can be leveraged to improve the quality of the control exerted by the MAPE loop [17].

Adaptation latency is considered for some very specific situations in some work. In the area of dynamic capacity management for data centers, the work of Gandhi et al. considers the setup time of servers, and is able to deal with unpredictable changes in load by being conservative about removing servers when the load goes down [20]. Their work is specifically tailored to adding and removing servers, a setting that resembles the example used in this paper. However, their work cannot reason about other tactics that could be used instead of or in combination with tactics to control the number of servers. Similarly, the work of Jamshidi et al. on autonomic scaling for cloud-based software using fuzzy logic for reasoning [25] does not consider the simultaneous use of tactics other than scaling. Zhang et al. propose a safe adaptation approach that can minimize the cost of adaptation, with adaptation duration being one such cost [46]. However, that cost is only considered once all the possible ways of reaching the desired target configuration have been found. That is, adaptation duration is not considered to select among alternative target configurations. In previous work, we presented an algorithm for proactive latency-aware adaptation based on dynamic programming [9]. That algorithm, however, did not deal adaptation under uncertainty.

8.3 Quantitative Verification

Calinescu et al. proposed the use of model checking and quantitative verification techniques at run time to ensure the dependability of self-adaptive systems [7]. In their approach, referred to as *runtime quantitative verification* (RQV), information gathered through the self-adaptive system's monitoring capability is used to update parameters in the formal model of the system, which is then used to detect or predict requirements violations. If a violation is detected, the same quantitative verification techniques can be used to select, for example, the configuration less likely to result in an unsatisfied requirement. Despite the use of a model checker at run time, Gerasimou et al. recently showed how the overhead and execution time of this approach can be reduced by combining caching, look-ahead, and near-optimal reconfiguration [22]. There are two main differences between that use of verification, and the use of probabilistic model checking in this paper. One is that for adaptation decisions, RQV is used to quantify or verify properties of each possible configuration one at a time, and that information is then used to select a target configuration outside of the model checking process. Instead, we use the model checker to synthesize the best adaptation strategy. The second difference is that they verify individual configurations in the context of a snapshot of the environment state, whereas we verify sequences of adaptations in the context of an evolving environment.

In previous work, we presented a technique to analyze different adaptation approaches using probabilistic model checking, and applied it to proactive latency-aware adaptation [9]. In that case, model checking was used as an offline analysis to study worst/best-case performance of an adaptation approach. The work presented here, in contrast, uses model checking at run time to decide *how* to adapt.

9. CONCLUSION

We have presented an approach for proactive latency-aware adaptation under uncertainty that uses probabilistic model checking to make adaptation decisions. The approach uses a look-ahead horizon to find the adaptation that maximizes the expected utility accumulated over the horizon in the context of the uncertainty of the environment. The advantages of using probabilistic model checking are that (i) the adaptation decision is optimal over the horizon because the model checker selects the strategy through a combination of mathematical models and exhaustive search; and (ii) it takes into account the stochastic behavior of the environment. Furthermore, the modular specification of tactics as separate processes, combined with the use of tactic compatibility predicates, allows the approach to deal easily with the (in)feasibility of adaptations due to the latency of tactics, and the conflicts (or lack thereof) between them. Our results showed that the approach performs better in terms of several metrics when compared with a feed-forward approach that does not use a look-ahead horizon and is not aware of tactic latency. Moreover, this advantage increases with the tactic latency.

Since the specification of tactics in the PRISM language is solely dependent on their applicability conditions and compatibility predicates, we plan in future work to generate them automatically from descriptions in a tactic specification language (e.g., an extension to Stitch [12]). We also intend to use such specification to determine whether tactics could interfere with each other, something we now assume is specified in the compatibility predicate. Additionally, we plan on analyzing the scalability of the approach for larger sets of adaptation tactics, as well as the use of optimization techniques to counter the effects of scale. Another open question worth investigating is how the period λ and the horizon length should be chosen, since these are likely dependent on the characteristics of the environment.

Even though in this paper we have focused on tactic latency as a key motivation for the approach, the same techniques would be useful for self-adaptive systems in which the use of a tactic prevents the use of other tactics in the near future.[9]

10. ACKNOWLEDGMENT

This work is supported in part by awards N000141310401 and N000141310171 from the Office of Naval Research (ONR), CNS 1116848 from the National Science Foundation, and by the National Security Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ONR or the U.S. government. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. (DM-0002240).

[9]For example, a small drone breaking from a formation would not be able to get shared location data from others in the formation, and that, in turn, would prevent the possibility of turning off its GPS to save power.

11. REFERENCES

[1] HAProxy: the reliable, high performance TCP/HTTP load balancer. `http://www.haproxy.org/`.

[2] RUBiS: Rice University Bidding System. `http://rubis.ow2.org/`.

[3] S. Andova, H. Hermanns, and J.-P. Katoen. Discrete-time rewards model-checked. In *FORMATS*, volume 2791 of *Lecture Notes in Computer Science*, pages 88–104. Springer, 2003.

[4] M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup web site. *IEEE Network*, 14(3):30–37, 2000.

[5] C. Bertolli, G. Mencagli, and M. Vanneschi. A cost model for autonomic reconfigurations in high-performance pervasive applications. In *Proceedings of the 4th ACM International Workshop on Context-Awareness for Self-Managing Systems - CASEMANS '10*, pages 20–29, New York, New York, USA, Sept. 2010. ACM Press.

[6] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. S. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science, 15th Conference, Bangalore, India, December 18-20, 1995, Proceedings*, volume 1026 of *Lecture Notes in Computer Science*, pages 499–513. Springer, 1995.

[7] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Communications of the ACM*, 55(9):69, Sept. 2012.

[8] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic QoS management and optimization in service-based systems. *IEEE Transactions on Software Engineering*, 37(3):387–409, May 2011.

[9] J. Cámara, G. A. Moreno, and D. Garlan. Stochastic game analysis and latency awareness for proactive self-adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems - SEAMS 2014*, pages 155–164, New York, New York, USA, June 2014. ACM.

[10] H. Chen and S. Hariri. An evaluation scheme of adaptive configuration techniques. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 493–496, Atlanta, Georgia, USA, 2007. ACM.

[11] B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. Software engineering for self-adaptive systems: A research roadmap. In B. H. C. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, June 2009.

[12] S.-W. Cheng and D. Garlan. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software*, 85(12):2860–2875, Dec. 2012.

[13] M. B. Dias, D. Locher, M. Li, W. El-Deredy, and P. J. Lisboa. The value of personalised recommender systems to e-business. In *Proceedings of the 2008 ACM Conference on Recommender Systems - RecSys '08*, page 291, New York, New York, USA, Oct. 2008. ACM.

[14] P. A. Dinda. Design, implementation, and performance of an extensible toolkit for resource prediction in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 17(2):160–173, Feb. 2006.

[15] S. Duttagupta, R. Virk, and M. Nambiar. Predicting performance in the presence of software and hardware resource bottlenecks. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2014)*, pages 542–549. IEEE, July 2014.

[16] N. Esfahani and S. Malek. Uncertainty in self-adaptive software systems. In R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 214–238. Springer Berlin Heidelberg, 2013.

[17] A. Gambi, D. Moldovan, G. Copil, H.-L. Truong, and S. Dustdar. On estimating actuation delays in elastic computing systems. In *2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 33–42. IEEE, May 2013.

[18] N. Gamez, L. Fuentes, and M. A. Aragüez. Autonomic computing driven by feature models and architecture in FamiWare. In *5th European Conference on Software Architecture*, pages 164–179, Essen, Germany, Sept. 2011. Springer-Verlag.

[19] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. Modeling the impact of workload on cloud resource scaling. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, pages 310–317. IEEE, Oct. 2014.

[20] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems*, 30(4), 2012.

[21] D. Garlan, B. Schmerl, and S.-W. Cheng. Software architecture-based self-adaptation. In Y. Zhang, L. T. Yang, and M. K. Denko, editors, *Autonomic Computing and Networking*, pages 31–55. Springer US, 2009.

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

[23] J. Hielscher, R. Kazhamiakin, A. Metzger, and M. Pistore. A framework for proactive self-adaptation of service-based applications based on online testing. In *1st European Conference on Towards a Service-Based Internet*, volume 5377, pages 122–133. Springer Berlin Heidelberg, 2008.

[24] M. Islam, S. Ren, H. Mahmud, and G. Quan. Online energy budgeting for cost minimization in virtualized

data center. *IEEE Transactions on Services Computing*, PP(99):1–1, 2015.

[25] P. Jamshidi, A. Ahmad, and C. Pahl. Autonomic resource provisioning for cloud-based software. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 95–104, Hyderabad, India, 2014. ACM.

[26] D. L. Keefer. Certainty equivalents for three-point discrete-distribution approximations. *Management Science*, 40(6):760–773, 1994.

[27] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[28] C. Klein, M. Maggio, K.-E. Å rzén, and F. Hernández-Rodriguez. Brownout: building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 700–711, New York, New York, USA, May 2014. ACM.

[29] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, Oct. 2014.

[30] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '02)*, pages 52–66. Springer-Verlag, 2002.

[31] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: verification of probabilistic real-time systems. In *23rd international conference on Computer Aided Verification*, pages 585–591. Springer-Verlag, July 2011.

[32] M. Z. Kwiatkowska and D. Parker. Automated verification and strategy synthesis for probabilistic systems. In D. V. Hung and M. Ogawa, editors, *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, volume 8172 of *Lecture Notes in Computer Science*, pages 5–22. Springer, 2013.

[33] P. Leitner, W. Hummer, and S. Dustdar. Cost-based optimization of service compositions. *IEEE Transactions on Services Computing*, 6(2):239–251, Apr. 2013.

[34] J. Liu, B. Priyantha, T. Hart, H. S. Ramos, A. A. F. Loureiro, and Q. Wang. Energy efficient GPS sensing with cloud offloading. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems - SenSys '12*, page 85, New York, New York, USA, Nov. 2012. ACM.

[35] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *10th USENIX Symposium on Networked Systems Design and Implementation*, pages 313–328. USENIX Association, Apr. 2013.

[36] L. Maatta, J. Suhonen, T. Laukkarinen, T. D. Hamalainen, and M. Hannikainen. Program image dissemination protocol for low-energy multihop wireless sensor networks. In *2010 International Symposium on System on Chip*, pages 133–138. IEEE, Sept. 2010.

[37] M. Mao and M. Humphrey. A performance study on the VM startup time in the cloud. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 423–430. IEEE, June 2012.

[38] J. A. Mccann and M. C. Huebscher. Evaluation issues in autonomic computing. In H. Jin, Y. Pan, N. Xiao, and J. Sun, editors, *Grid and Cooperative Computing*, volume 3252 of *Lecture Notes in Computer Science*, pages 597–608, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[39] A. Metzger, O. Sammodi, and K. Pohl. Accurate proactive adaptation of service-oriented systems. In J. Cámara, R. de Lemos, C. Ghezzi, and A. Lopes, editors, *Assurances for Self-Adaptive Systems*, volume 7740, pages 240–265. Springer Berlin Heidelberg, 2013.

[40] I. D. Paez Anaya, V. Simko, J. Bourcier, N. Plouzeau, and J.-M. Jézéquel. A prediction-driven adaptation approach for self-adaptive sensor networks. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 145–154. ACM, 2014.

[41] M. Parashar and S. Hariri. Autonomic computing: An overview. In J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, editors, *Unconventional Programming Paradigms*, Lecture Notes in Computer Science, pages 257–269. Springer Berlin Heidelberg, 2005.

[42] V. Poladian, D. Garlan, M. Shaw, M. Satyanarayanan, B. Schmerl, and J. Sousa. Leveraging resource prediction for anticipatory dynamic configuration. In *Self-Adaptive and Self-Organizing Systems*, pages 214–223. IEEE, July 2007.

[43] K. Qazi, Y. Li, and A. Sohn. Workload prediction of virtual machines for harnessing data center resources. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 522–529. IEEE, June 2014.

[44] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):1–42, May 2009.

[45] C. Wang and J.-L. Pazat. A two-phase online prediction approach for accurate and timely adaptation decision. In *2012 IEEE Ninth International Conference on Services Computing*, pages 218–225. IEEE, June 2012.

[46] J. Zhang, Z. Yang, B. H. C. Cheng, and P. K. McKinley. Adding safeness to dynamic adaptation techniques. In *Proceedings of the ICSE 2004 Workshop on Architecting Dependable Systems*, Edinburgh, Scotland, 2004.

[47] J. Zhang and B. Zwart. Steady state approximations of limited processor sharing queues in heavy traffic. *Queueing Systems*, 60(3-4):227–246, Nov. 2008.