Efficient Decision-Making under Uncertainty for Proactive Self-Adaptation

Gabriel A. Moreno*†, Javier Cámara†, David Garlan† and Bradley Schmerl†
*Software Engineering Institute, †School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
*gmoreno@sei.cmu.edu, †{jcmoreno,garlan,schmerl}@cs.cmu.edu

***Abstract*—Proactive latency-aware adaptation is an approach for self-adaptive systems that improves over reactive adaptation by considering both the current and anticipated adaptation needs of the system, and taking into account the latency of adaptation tactics so that they can be started with the necessary lead time. Making an adaptation decision with these characteristics requires solving an optimization problem to select the adaptation path that maximizes an objective function over a finite look-ahead horizon. Since this is a problem of selecting adaptation actions in the context of the probabilistic behavior of the environment, Markov decision processes (MDP) are a suitable approach. However, given all the possible interactions between the different and possibly concurrent adaptation tactics, the system, and the environment, constructing the MDP is a complex task. Probabilistic model checking can be used to deal with this problem since it takes as input a formal specification of the stochastic system, which is internally translated into an MDP, and solved. One drawback of this solution is that the MDP has to be constructed every time an adaptation decision has to be made to incorporate the latest predictions of the environment behavior. In this paper we present an approach that eliminates that run-time overhead by constructing most of the MDP offline, also using formal specification. At run time, the adaptation decision is made by solving the MDP through stochastic dynamic programming, weaving in the stochastic environment model as the solution is computed. Our experimental results show that this approach reduces the adaptation decision time by an order of magnitude compared to the probabilistic model checking approach, while producing the same results.**

I. Introduction

Self-adaptive systems have mechanisms to change their structure and/or behavior in order to deal with changes in their environment, such as workload and resource fluctuations, and security threats [1]–[3]. Most self-adaptation approaches are reactive, making adaptation decisions based on current conditions [4]. Unless there is an adaptation cost, being reactive is not a problem if the system can adapt very quickly, because at any point, the system can rapidly change to best deal with the conditions at that moment. However, not all adaptation tactics are instantaneous. For example, provisioning a new virtual machine in the cloud can take a few minutes [5]. We refer to the period of time between when a tactic is started and when its effect is produced as *tactic latency*. The problem with tactics that have non-trivial latency is that not all system configurations are possible at all times. For instance, if adding a new server to a system takes two minutes, it is not possible to reach a system configuration with one more server in one minute. The only way to have that additional server on time is to start its addition *proactively*, taking into account the latency of that tactic.

Tactic latency also matters when the system can use tactics with different latencies to deal with the same situation. For example, an alternative to adding capacity with a new server, is to reduce load by reducing the quality of service (QoS); something that can be done with a much faster tactic. In a situation like this, considering not only the effect of the tactics on the system, but also their latency when deciding how to adapt can result in more effective adaptations.

Latency-awareness is even more useful when concurrent tactic execution is supported. In that case, it is possible to complement slow tactics with fast ones if they do not interfere with each other. For example, suppose at some point the tactic to add a server is started because that was deemed appropriate to handle a predicted increase in the request rate to the system. However, the next time the system evaluates its state—but before the tactic to add a server completes—the request rate is worse than was estimated. In this case, the system can reduce the QoS—and the load—right away using a fast tactic.

Another effect of tactic latency is that the execution of a tactic with considerable latency can prevent the use of other incompatible tactics while it executes (e.g., removing a server while it is being added). Consequently, an adaptation choice made at some point constrains the possible adaptations in subsequent decisions.

Proactive latency-aware adaptation is an approach that improves over reactive adaptation by considering both the current and anticipated adaptation needs of the system, and taking into account the latency of adaptation tactics [6], [7]. Making an adaptation decision with these characteristics requires solving an optimization problem to select the adaptation path that maximizes an objective function over a finite look-ahead horizon. This requires relying on predictions of the state of the environment over the decision horizon, which are not perfect and have uncertainty. Since this is a problem of selecting adaptation actions in the context of the probabilistic behavior of the environment, Markov decision processes (MDP) are a suitable approach. However, given all the possible interactions between the different and possibly concurrent adaptation tactics, the system, and the environment, constructing the MDP is a complex task.

978-1-5090-1654-9/16 $31.00 © 2016 IEEE
DOI 10.1109/ICAC.2016.59

In previous work, we proposed a proactive latency-aware approach to self-adaptation that uses probabilistic model checking to deal with this problem [7]. The probabilistic model checker takes as input a formal specification of the adaptive system and its stochastic environment, which is internally translated into an MDP, and solved. The solution to the MDP is the set of tactics that have to be started in order to achieve the adaptation goal (e.g., utility maximization). Using MDPs in this way, it is possible to reason about latency and uncertainty. However, the probabilistic transitions of the MDP depend on the stochastic behavior of the environment, which can only be estimated at run time, and with a short horizon. Consequently, the high overhead of constructing the MDP must be incurred every time an adaptation decision has to be made, so that the latest predictions of the environment behavior can be incorporated.

In this paper we present an approach that practically eliminates the run-time overhead of constructing the MDP by doing most of that *offline*. Using formal methods, the approach exhaustively considers the many possible system states, and combinations of tactics, including their concurrent execution when possible. At run time, the adaptation decision is made by solving the MDP through stochastic dynamic programming, weaving in the stochastic environment model as the solution is computed. To explain and evaluate the approach, we use RUBiS, a web application that has the core functionality of an auctions website [8]. Our experimental results show that this approach reduces the adaptation decision time by an order of magnitude compared to the probabilistic model checking approach, while still producing the same results.

The rest of the paper is organized as follows. The example that is used throughout the paper is presented in Section II. In Section III, we describe the general adaptation model within which the adaptation decision approach proposed in the paper works. The core of the approach is presented in Section IV. The evaluation of the approach is described in Section V, including the results. Related work is presented in Section VI. Our conclusions and future work directions are in Section VII.

II. Example

To illustrate and validate the approach, we use RUBiS, an open-source benchmark application that implements the functionality of an auctions website [8]. This application is widely used for research in web application performance, and various areas of cloud computing [9]–[12]. RUBiS is a multi-tier web application consisting of a web server tier that receives requests from clients using browsers, and a database tier. In our setup, we also include a load balancer to support multiple servers in the web tier. The load balancer distributes the requests among the servers following a round-robin policy. When a client requests a web page, the web server accesses the database tier to get the data needed to render the page with the dynamic content. The request arrival rate, which induces the workload on the system, changes over time. This changing arrival rate is the only relevant property of the system's environment that is considered for this example.

RUBiS was not designed as a self-adaptive system, but we added an adaptation layer to make it self-adaptive. We included two pairs of inverse adaptation tactics that can be used to deal with the changing arrival rate and the load it induces. One pair of tactics can be used to add and remove servers, thus changing the capacity of the system. The tactic to add a server has a latency λ.[1] The inverse tactic removes a server. Although this requires waiting for the server to complete processing the requests being handled by the server, we assume that time to be negligible, and thus assume the tactic to be immediate.[2] The other pair of tactics leverages the *brownout* paradigm [13]. 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 related items. A parameter called *dimmer* controls the proportion of responses that include the optional content. In that way, it is possible to use the dimmer to control the load on the system. The value of the dimmer can be thought of as the probability of a response including the optional content, thus taking values in $[0..1]$. To control the dimmer, the system has two immediate adaptation tactics that increase and decrease its value. We allow tactics to be executed concurrently only if they belong to different pairs. For example, if a server is being added, a server cannot be removed, but it is possible to increase or decrease the dimmer.

The goal of self-adaptation in our example is to maximize the utility provided by the system at the minimum cost. The utility is computed according to a service level agreement (SLA) with rewards for meeting the target average response time over a measurement interval, and penalties when the response time is not met [14]. The cost is proportional to the number of servers used. The SLA specifies a target response time T. The utility obtained in an interval depends on whether the target is met or not, as given by

$$U = \begin{cases} \tau a(dR_O + (1-d)R_M) & \text{if } r \leq T \\ \tau \min(0, a-\kappa)R_O & \text{if } r > T \end{cases} \quad (1)$$

where τ is the length of the interval, a is the average request rate, r is the average response time, d is the dimmer value, κ is the maximum request rate the site is expected to handle, and R_M and R_O are the rewards for serving a request with mandatory and optional content, respectively, with $R_O > R_M$.

III. Self-Adaptation Model

Our approach fits in the general class of self-adaptation architectures based on explicit closed-loop control such as the MAPE-K autonomic manager [15]. The MAPE phases cover the activities performed in the control loop: (i) monitoring the system and the environment; (ii) analyzing the information collected and deciding if adaptation is needed; (iii) planning how to adapt; and (iv) executing the adaptation. These activities share a knowledge model that integrates them.

[1]The latency is assumed to be constant, but if it was a random variable, λ would be its expected value.

[2]This is just a choice we made for this example. If that time was not negligible, the tactic could be modeled as a tactic with latency.

Even though MAPE-K has distinct analysis and planning phases, these are combined into a single activity in our approach because when the goal is to maximize a utility function, determining whether the system can adapt to a configuration that will give higher utility (the analysis part) implies finding such configuration (the planning part). The adaptation decision phase is run periodically, at a fixed interval τ, to determine both whether adaptation is required, and what tactics to use.

IV. Adaptation Decision

The goal of the adaptation decision is to determine what adaptation action to take, if any, with the goal of maximizing the utility that the system will provide over the rest of its execution. In principle, each adaptation decision could be made in isolation. However, when adaptations have latency, reacting to the current situation without looking ahead can result in suboptimal decisions even if there is no adaptation cost [7]. The reason is that the configuration of the system at any given time constrains the possible configurations at a later time. 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.

Considering that the decision made at a given time affects the possible evolutions of the system and constraints subsequent decisions, the adaptation decision problem is a sequential decision problem [16]. The managed system and its environment evolve through time, and the adaptation manager has to decide at regular intervals what adaptation action to take, if any, to maximize the utility that the system will provide. Since the utility the system provides with a given configuration depends on the state of the environment, the decision process must be based on the joint process that describes the combined behavior of the system and its environment. In this work we assume that the effect of tactics on the system configuration is deterministic, and depends only on the current state of the system. However, the behavior of the environment is stochastic, thus making the transitions in the joint system/environment process probabilistic. Consequently, the adaptation decision problem can be formulated as a Markov decision process.

The stochastic model of the environment in the MDP is based on predictions of the future state of the environment. In our example, it is built using a time series predictor. Taking into account that the uncertainty of these predictions increases as they get further into the future, it is not practical to look too far ahead. Therefore, the adaptation decision is formulated as a discrete-time sequential decision problem with finite horizon, and its solution approximates the original decision problem, determining what adaptation tactics should be started at the current time, if any, to maximize the aggregate utility the system will provide over the decision horizon.

A. Adaptation Decision Problem Formulation

A new adaptation decision is made at regular intervals of length τ, and each decision itself is the solution of a discrete-time finite horizon decision problem, in which time is discretized into intervals of length τ, with a horizon of H intervals. The adaptation decision considers two kinds of configuration changes or transitions due to adaptation: immediate, and delayed. Immediate transitions are the result of either the execution of tactics with very low latency (e.g., changing the dimmer value), or the start of a tactic with latency (e.g., adding a server). In the latter case, the transition is immediate because the target state is a configuration in which a new server is being added, but the addition has not been completed yet. Delayed configuration changes are due to adaptation as well, but also require the passing of time for the transition to happen. This is the case, for example, with the addition of a new server, transitioning from a state in which the server is being added to a state in which the addition has completed.

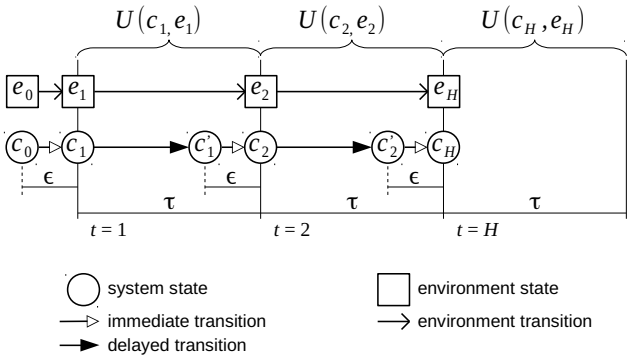

Fig. 1. Pattern of adaptation transitions in adaptation decision solution

Any solution to the adaptation decision problem will follow the same pattern of immediate and delayed transitions as shown in Fig. 1 for $H = 3$. The interval $t = 1$ corresponds to the interval of length τ starting at the current time, interval $t = 2$ starts τ later, and so on. To simplify the presentation, we start with two assumptions that will be relaxed later: (i) the evolution of the environment over the decision horizon is known deterministically; and (ii) each tactic is either instantaneous or its latency is approximately τ. The state of the environment in interval t is e_t.[3] State c_0 represents the configuration of the system when the adaptation decision is being made. At that point, an immediate transition takes the system from c_0 to c_1 right before the first interval starts. The negligible time that this transition takes is denoted as ϵ, and it is shown disproportionately large in the figure so that it can be drawn. The passage of time causes the configuration to change from c_1 to c'_1. For example, if c_1 is a configuration in which the addition of a server has been started, c'_1 is one with the server addition completed. After that, another immediate transition resulting in c_2 takes place, then the second interval starts, and so on. For the purpose of considering the utility accumulated over the decision horizon, the intermediate configuration that

[3]In general, the state of the environment can change during a decision interval. However, for the decision problem, a metric representative of the state throughout the interval is used (e.g., the average request arrival rate).

precedes each immediate transition is ignored, and the utility accrued in interval t is $U(c_t, e_t)$.

In general decision problems, the solution is found by considering all the actions that are applicable in each state, and the result is a policy that maps states to actions. However, in our setting there are two reasons why finding and expressing the solution directly in terms of actions is not practical. First, our approach supports concurrent execution of tactics, which means that more than one tactic (or action) can be started simultaneously, resulting in a single transition to a configuration with the combined effect of the tactics. Second, there can be tactics with latency longer than the decision interval, which means that once the tactic has started, it is possible to have transitions that are exclusively due to the passage of time. Instead of dealing directly with actions, we use predicates over pairs of states that indicate whether configuration c' can be reached from configuration c. These reachability predicates are:

- $R^I(c, c')$, which is true if configuration c' can be reached with an immediate transition from c with the use of none, one or more tactics; and
- $R^D(c, c')$, which is true if configuration c' can be reached with a delayed transition from c in one time interval.

A third helper predicate, used for a more compact notation, is true if c' can be reached from c in one time interval through a delayed transition followed by an immediate transition:

$$R^T(c, c') \equiv \exists c'' : R^D(c, c'') \wedge R^I(c'', c')$$

Defining these predicates is not trivial due to the possible interactions between different tactics, which requires exploring all the possible combinations of tactics. In our approach, we use formal methods to compute these predicates offline (as it is explained in Section IV-B), reducing the burden on the run-time decision algorithm.

These predicates define the transition matrix for the system portion of the adaptation MDP. Therefore, a solution like the one shown in Fig. 1 is feasible only if $R^I(c_0, c_1)$ and $R^T(c_t, c_{t+1}), \forall t = 1, \ldots, H-1$ hold. To find the solution, let us refer to the set of all system configurations as C. This set contains all the configurations that are unique with respect to the properties relevant to computing the utility function. In our example, these properties include the number of active servers, and the dimmer value. Later on, this set will be extended to capture the state of running tactics in the system configuration. Let us also define sets of configurations that can be reached from a given configuration using different kinds of transitions:

$$C^T(c) = \{c' \in C : R^T(c, c')\}$$
$$C^I(c) = \{c' \in C : R^I(c, c')\}$$

With the assumption of a deterministic environment, the solution C^* to the adaptation decision problem can be found using dynamic programming as follows:

$$v^H(c) = \hat{U}(c, e_H), \quad \forall c \in C \tag{2}$$

$$v^t(c) = \hat{U}(c, e_t) + \max_{c' \in C^T(c)} v^{t+1}(c'), \quad t = H-1, \ldots, 1 \tag{3}$$

$$C^* = \arg\max_{c' \in C^I(c_0)} v^1(c') \tag{4}$$

Note that the utility function used to solve the adaptation decision problem is the decision utility function $\hat{U}$, which has some differences with respect to U in (1), the one used to measure the utility of the system. The first difference is that the response time used in the computation is not the measured response time of the system, since $\hat{U}$ is used to compute the utility that the system would attain under a certain configuration and state of the environment. Therefore, the response time has to be estimated. In our case, we resort to queueing theory using a limited processor sharing (LPS) model, which models a system in which the number of concurrent requests that can be processed simultaneously by each server is limited by a constant [17]. For the web servers in our example, this constant is equal to the maximum number of processes configured for them. Another difference is that for our particular self-adaptation goal, $\hat{U}$ is defined so that if $U(c_1, e) = U(c_2, e)$, then $\hat{U}(c_1, e) > \hat{U}(c_2, e)$ if c_1 has lower cost (this is achieved by scaling the original utility values).[4]

The result of $\arg\max$ in (4) is actually a set, so we can pick any configuration $c^* \in C^*$. However if $c_0 \in C^*$, we can avoid adapting, since no configuration change would render any improvement. Since the actions in our setting are deterministic, given the source and target of a transition, it is possible to determine the actions that have to be taken as it will be explained later. Therefore, once c^* is found, the set of tactics that have to be started to reach it from c_0 can be determined.

1) Stochastic Environment: We model the evolution of the environment over the decision horizon as a discrete-time Markov chain. The set of environment states is denoted by E, and the probability of transitioning from state e to state e' is given by $p(e'|e)$. Although how the set E and the transition probabilities for the environment are obtained is not relevant, for this work we used a time series predictor to predict future request rates based on past observations of the environment. These predictions and their uncertainty are then used to generate a probability tree using the technique described in our previous work [7].

The most straightforward way to take into account the stochastic evolution of the environment would be to create the joint MDP of the system MDP and the environment Markov chain,[5] and then find its solution. However, this would require creating the transition probability matrix over the joint state space $C \times E$, and evaluating many joint states that would

[4] In a case in which all the configurations would exceed the target response time, the utility function would choose the one with the smallest number of servers. This is not the right decision because removing resources from an overloaded system would cause the backlog of requests to increase at a higher rate, making the recovery of the system in subsequent decisions even more unlikely. Therefore, an exception to this rule is included in $\hat{U}$ to favor the configuration with the most servers in such a case.

[5] A discrete-time Markov chain can be turned into an MDP by assuming there is a single action applicable in every state.

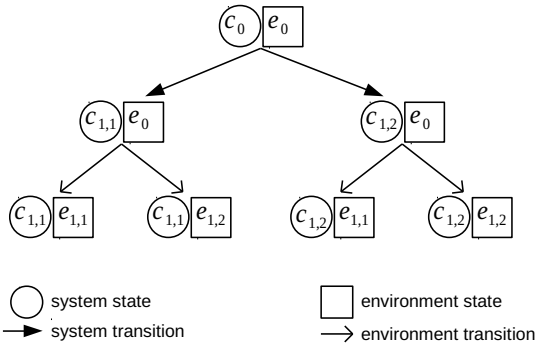

Fig. 2. System and environment transitions

never be reachable. Keeping in mind that this would have to be done every time an adaptation decision has to be made, doing this has a couple of drawbacks. First, the full joint MDP would have to be created for every decision so that the latest environment predictions could be incorporated. Second, evaluating the utility for a pair of system configuration and environment state may involve some extensive computation, so doing that for unreachable joint states is a waste of resources and time.

To reduce the running time of the adaptation decision, we avoid creating the joint MDP, and instead weave the environment model into the predefined MDP of the system as needed. Referring to Fig. 1, we can see that the system and the environment make a transition almost simultaneously at the beginning of each interval. For example, at the start of the interval at $t = 1$, the system transitions from c_0 to c_1 (the alternative target configurations are not shown), and the environment transitions from its current state e_0 to e_1. The difference with the stochastic environment is that both the system and the environment have several possible target states at each interval. Fig. 2 depicts these two kinds of transitions interleaved. First, the system takes a deterministic transition, and then the environment takes a probabilistic transition.

Using the principles of stochastic dynamic programming [16], the adaptation decision problem with a stochastic model of the environment can be solved as follows:

$$v^H(c,e) = \hat{U}(c,e), \quad \forall c \in C, e \in E_H \tag{5}$$

$$v^t(c,e) = \hat{U}(c,e) + \max_{c' \in C_T(c)} \sum_{e' \in E_{t+1}} p(e'|e) v^{t+1}(c',e'), \quad t = H-1,\ldots,1 \tag{6}$$

$$C^* = \arg\max_{c' \in C_I(c_0)} \sum_{e' \in E_1} p(e'|e_0) v^1(c',e') \tag{7}$$

Note that instead of evaluating all environment states in E for each time interval, only those that are feasible are considered. E_t is the set of environment states feasible in time interval t. When a probability tree is used to model the environment, E_t is the set of nodes of depth t.

2) Handling Latency: When a tactic has latency, the adaptation decision has to be able to determine when the tactic is going to complete, so that its effect on the system configuration can be accounted for at the right time. In addition, while a tactic executes, it can prevent other incompatible tactics from starting, which affects the decision. So far, we assumed that if a tactic had latency, it was roughly equal to one time interval, but in reality it can be of any length, and span multiple intervals. Since we use a Markov model, there is no history in the model to allow us to directly keep track of the progress of the tactic. Consequently, we have to extend the state space in the model to keep track of the progress of tactics with latency. However, given that decisions are made at regular intervals over the decision horizon, it is necessary to keep track of the progress of the tactic only at the granularity of the time interval.

For our running example, the configuration of the system with the properties relevant for computing the utility function can be captured by a tuple (s,d), where s is the number of active servers, and d is the discretized dimmer value. In order to keep track of the progress of the tactic to add a server, we extend the configuration with another component, so that the full configuration tuple is (s,d,p_{add}), where $p_{add} \in \left\{0,\ldots,\lceil\frac{\lambda}{\tau}\rceil\right\}$ is the number of time intervals left until the tactic completes, with 0 indicating that the tactic is not being executed.[6] For example, the start of the tactic to add a server implies an immediate transition in the model to a configuration with p_{add} equal to its maximum value. This transition is enabled by R^I. It is then be followed by a sequence of delayed transitions enabled by R^D that decrease the value of p_{add} until it reaches 0, and when that happens, the number of servers in the configuration tuple is increased.

B. Computing Reachability Predicates

The predicates R^I and R^D determine which system configurations can be reached from other configurations through adaptation; that is, they specify which transitions are feasible in the system MDP. Defining these predicates by extension, or trying to express them in propositional logic can be a daunting and error prone task due to all the possible combinations of tactics, all their possible phasings (i.e., how their executions overlap in time), and all the possible system states that must be taken into account. Instead, we use formal methods to compute the reachability predicates. Specifically, we use Alloy [18] to formally specify system configurations and adaptation tactics, and to compute the reachability predicates. Alloy is a language based on first-order logic that allows modeling structures—known as signatures—and relationships between them in the form of constraints. Alloy is a declarative language, and, in contrast to imperative languages, only the effect of operations—tactics in our case—on the model must be specified, but not how the operations work. The Alloy analyzer is used to find structures that satisfy the model. Thanks to delaying as much as possible combining the system and environment states in our approach,

[6]In our example, each tactic cannot execute concurrently with itself, so a single value can track its progress. If multiple instances of a tactic could be executed concurrently, one progress component per instance would be needed.

```
1  open util/ordering[S] as SO
2  open util/ordering[D] as DO
3  sig S {} // the different number of active servers
4  sig D {} // the different dimmer levels
5
6  // each element of C represents a configuration
7  sig C {
8      s : S, // the number of active servers
9      d : D // dimmer level
10 }
```

Fig. 3. Alloy model: configurations

these predicates are independent of the environment state, and thus can be computed offline. Hence, the overhead of using formal methods to compute the predicates is not incurred at run time.

The support for concurrent tactics in the adaptation decision is handled by these predicates. The adaptation problem formulation (5)-(7) is agnostic with regard to concurrent tactic execution, since it only cares about state reachability, regardless of whether that requires concurrent tactics or not. On the other hand, to correctly determine whether a configuration can be reached from another configuration when computing the predicates, it is necessary to consider whether tactics can be executed concurrently or not. To that end, we rely on a *compatibility predicate* for each tactic that indicates whether it can be run, considering the other tactics that are executing. We require that two tactics are allowed to execute concurrently only if they affect disjoint subsets of the properties of the configuration state. From the formal model perspective, this requirement makes the tactics serializable, since the state resulting from their serial application would be the same as if they were applied in parallel.

To compute R^D, we use Alloy to find all the pairs $(c, c') \in \text{CP} \times \text{CP}$, such that $R^D(c, c')$, where CP is the configuration space extended with tactic progress. To achieve that, we introduce first other necessary pieces of the model. Fig. 3 shows the declarations that define the system configuration space for our example. The sets `S`, and `D` represent the different numbers of active servers, and dimmer levels, respectively. The elements of these sets are not numbers, but just abstract elements. However, lines 1-2 specify that these are ordered sets. Thus, we can refer to their first and last elements, for example, with `SO/first` and `SO/last`. Also, we can get the successor and predecessor of an element `e` with `SO/next[e]` and `SO/prev[e]`. The signature `C` defines the set of all possible configurations, each having a number of active servers `s`, and a dimmer level `d`. In the Alloy model, we distinguish between plain system configurations, `C`, and configurations extended with tactic progress, `CP`. The reason we do this is for the code to be more modular, keeping concerns separated, so that it is easier to generate the Alloy code for different systems, and/or different sets of tactics. Note, however, that when the adaptation decision problem is solved, `CP` in this model corresponds to C in the formulation presented in Section IV-A.

Fig. 4 shows the elements needed to represent tactic progress. The declaration of the set of all the tactics `T` as *abstract* in line 3 indicates that all its elements must be elements of one of the signatures that extends it. For each of the tactics with no latency, a singleton set extending `T` is declared (line 4). Since it is necessary to tell tactics with latency apart, the abstract subset `LT` is declared (line 5), and a singleton subset of it is declared for each of the tactics with latency (line 6, only `AddServer` in our example). The different levels of progress of each tactic are represented by the elements of an ordered set. For example, `TPAS` (lines 1 and 8) contains the levels of progress of the tactic to add a server, with `TPASO/first` indicating the tactic has just started, `TPASO/last` indicating that the tactic execution has completed, and the elements in between representing intermediate progress. The ordered sets that represent the levels of progress of tactics are subsets of an abstract set `TP`. The signature `CP` extends `C`, adding a mapping from tactics with latency, `LT`, to the tactic progress, `TP`. The facts in lines 14-15 constrain `p` to be a function over `LT`. Additionally, we require that the function maps each tactic to a progress in its corresponding class (line 16). Lastly, the fact in line 19 requires that all elements of `CP` are different.

```
1  open util/ordering[TPAS] as TPASO // tactic progress for adding server
2
3  abstract sig T {} // all tactics
4  one sig IncDimmer, DecDimmer, RemoveServer extends T {} // tactics
       with no latency
5  abstract sig LT extends T {} // tactics with latency
6  one sig AddServer extends LT {} // tactic with latency
7  abstract sig TP {} // tactic progress
8  sig TPAS extends TP {} // one sig for each tactic with latency
9
10 // configuration extended with the progress of each tactic with latency
11 sig CP extends C {
12     p: LT -> TP
13 } {
14     ~p.p in iden // p maps each tactic to at most one progress
15     p.univ = LT // every tactic in LT has a mapping in p
16     p[AddServer] in TPAS // restrict each tactic to its own progress class
17 }
18
19 fact uniqueConfigs { all disj c1, c2 : CP | !equals[c1, c2] or c1.p != c2.p}
```

Fig. 4. Alloy model: configurations extended with tactic progress

Now that we have all the basic elements in the model, we can present the predicates that determine the reachability in one time interval. For each tactic with latency, a predicate like `addServerTacticProgress`, shown in Fig. 5, is needed. This predicate is true if according to the tactic, the post-state `c'` can be reached in one time interval from the pre-state `c`. If the tactic is running, the predicate requires that in the post-state, the progress of the tactic is the next one. If it reaches the last level of progress, then the configuration has one more server in the post-state (lines 3-4), reflecting the effect of the completion of the tactic. In addition, it is as important to ensure that if the tactic is not running, it stays in that state (line 6), and does not have an effect (line 7). We also need to require that nothing else changes (lines 10-11). Finally, the predicates for the progress of each tactic with latency have to be put together to define `progress`, a predicate equivalent to R^D (line 14-16). If we had more than one tactic with latency, their predicates would have to be composed to reflect the effect that

```
pred addServerTacticProgress[c, c' : CP] {
    c.p[AddServer] != TPASO/last implies { // tactic is running
        c'.p[AddServer] = TPASO/next[c.p[AddServer]]
        c'.p[AddServer] = TPASO/last implies c'.s = SO/next[c.s] else c'.s =
            c.s
    } else {
        c'.p[AddServer] = TPASO/last // stay in not running state
        c'.s = c.s
    }
    // nothing else changes other than s and the progress of this tactic
    equalsExcept[c, c', C$s]
    (LT − AddServer) <: c.p in c'.p
}

pred progress[c, c' : CP] {
    addServerTacticProgress[c, c']
}
```

Fig. 5. Alloy model: tactic progress predicate

all of them would have on the state. All the progress predicates are serializable, because they either correspond to tactics that can execute concurrently, for which serializability is required; or they correspond to incompatible tactics. In the latter case, only one of them could be in a state in which it can affect the configuration, whereas the rest would have no effect, making them serializable as well. Therefore, all the progress predicates can be combined using sequential composition [19].

With the complete model, the Alloy analyzer is run to find all the instances that satisfy `progress`. Alloy requires that a scope (i.e., cardinality, either exact or as a bound) be provided for the different sets in the model. In our case the scope can be determined based on the maximum number of servers for the system, the number of dimmer levels, and the number of time intervals needed for the execution of tactics with latency. The output of Alloy can be read using its API, and used to generate a simple encoding of R^D as a lookup table suitable for use at run time when a decision has to be made.

In order to compute R^I, we define a predicate for each tactic that checks whether the tactic is applicable, and if so, it reflects the effect of the tactic on the post-state. However, we cannot simply compose them sequentially, as we do for R^D, because it is necessary to consider cases in which a tactic is not used even if its applicable. That is, we need to consider every possible combination of tactics that can be applied concurrently. The approach we take to deal with this problem is to model a trace of configuration states such that each element of the trace is related to its predecessor by either the application of a tactic, or the identity relation. Using the Alloy analyzer we can find all the possible traces that satisfy this model, and the set of all the pairs formed by the first and last state of each trace is the relation R^I.

Fig. 6 shows a portion of the model to compute R^I. In addition to computing R^I, we also need to compute for each pair in that relation the (possibly empty) set of tactics that have to be started for the immediate transition represented by the pair to hold. This is used to determine which tactics have to be started once the solution to (7) is found. To accomplish that, the elements of the trace have not only the configuration state, but also a set of tactics that have been started to arrive

```
1  open util/ordering[TraceElement] as Trace
2
3  sig TraceElement {
4      cp : CP,
5      starts : set T // tactic started
6  }
7
8  pred addServerCompatible[e : TraceElement] {
9      e.cp.p[AddServer] = TPASO/last
10     !(RemoveServer in e.starts)
11 }
12
13 pred addServerTacticStart[e, e' : TraceElement] {
14     addServerCompatible[e] and e.cp.s != SO/last
15     e'.starts = e.starts + AddServer
16     let c = e.cp, c'=e'.cp | {
17         c'.p[AddServer] = TPASO/first
18         // nothing else changes
19         equals[c, c']
20         (LT − AddServer) <: c.p in c'.p
21     }
22 }
23
24 fact traces {
25     let fst = Trace/first | fst.starts = none
26     all e : TraceElement − last | let e' = next[e] | {
27         equals[e, e'] and equals[e', Trace/last]
28     } or addServerTacticStart[e, e'] or removeServerTactic[e, e'] or
           decDimmerTactic[e, e'] or incDimmerTactic[e, e']
29 }
```

Fig. 6. Alloy model: predicates for tactic start

at that particular state in the trace (lines 3-6).

For each tactic with latency there is a predicate that models the start (but not the effect, which is delayed) of the tactic (line 13-22). It first checks that the tactic is compatible and applicable in the pre-state (line 14). In the post-state, the tactic is added to the set of tactics started (line 15), and the progress of the tactic is set to the first level (line 17). In addition, the predicate ensures that nothing else changes (lines 19-20). For instantaneous tactics, the predicate follows the same pattern, except that the effect of the tactic on the post-state is included (e.g., an increase of the dimmer value), and no tactic progress state is affected. The fact `traces` defines what a valid trace is. Line 25 states that at the beginning of the trace no tactic has been started at this time. The remainder of the fact specifies that every trace element is related to its predecessor by one of the predicates for the tactics, or is the same as its predecessor.

Similarly to what is done for computing R^D, Alloy is used to find all the possible instances that satisfy the model, and through its API we can obtain all the traces needed to construct R^I as a lookup table. Also, a map that associates pairs in R^I to the set of tactic starts is constructed to be used at run time.

V. Evaluation

The objectives of the evaluation were to assess the adaptation decision speedup attained by the proposed approach; and to confirm that the effectiveness of the adaptation decision was not affected, maintaining the advantage of proactive latency-aware adaptation. To this end, the experiments were done using three different adaptation approaches. The first proactive latency-aware approach, PLA-SDP, is the one presented in this paper. The second one, PLA-PMC, corresponds to the approach based on probabilistic model checking [7]. The

adaptation decision in PLA-PMC uses PRISM version 4.3 [20] to build and solve the MDP at run time. The third adaptation approach is a feed-forward (FF) approach that is latency-agnostic. It uses a one-step-ahead prediction of the request rate to select the adaptation tactic that would result in the highest utility, assuming that tactics are instantaneous, and not looking beyond the current decision.

For the evaluation, the system was deployed on a quad-core server running Ubuntu Server 14.04 as the host OS, with three virtual machines (VM), also running Ubuntu, each pinned to a dedicated core. These cores were isolated, and thus not used by the host OS. The VMs were used to deploy up to three web servers with RUBiS. The version of RUBiS we used was one modified by Klein et al. to support brownout [13]. The load balancer HAProxy [21] was run in the host OS to distribute requests among the servers. In order to keep the latency of the tactic to add a server experimentally controlled, the server VMs were kept running at all times, and the addition and removal of a server was simulated by enabling and disabling the server in the load balancer, respectively. When the tactic to add a server was used, the execution manager enabled the server in the load balancer after a time of λ had elapsed, simulating the latency of the tactic. The adaptation layer (monitoring, adaptation decision, execution manager, and knowledge model) was also deployed in the host OS, and a second computer was used to generate traffic to the website.

The period for the adaptation layer (i.e., the monitoring and adaptation interval) was $\tau = 60$ seconds. The length of the look-ahead horizon used for the adaptation decision was computed 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. In this way, the horizon is long enough for the system to go from one server to S_{max}, with an additional time interval to observe the benefit. A minimum of 5 intervals enforces look-ahead even if the tactic latency is small. The parameters of the utility function were set as follows: target response time $T = 0.75$ seconds; rewards for responses with mandatory and optional $R_M = 1$, $R_O = 1.5$ respectively; and maximum system capacity $\kappa = 67.4$ requests per second (this value was obtained through profiling). The adaptation tactics could change the number of servers between 1 and S_{max}, and the dimmer among the values 0.10, 0.30, 0.50, 0.70, and 0.90.

The stream of requests to the system was generated from publicly available traces captured from real websites. Specifically, we used half day from WorldCup '98 trace [22], and one day from the ClarkNet trace [23].[7] Both traces were scaled to last for 105 minutes, and to reach the maximum capacity of the validation setup at their peak. They were replayed using a client able to make as many concurrent requests as needed to reproduce the requests according to their timestamps. All the requests targeted a single URL, which in turn selected a random item from the auction to render its details page.

We compared the PLA approaches against FF to assess the

[7]The point of using these traces is to exercise the system with realistic traffic patterns, and not to replicate the behavior of users of an auctions website.

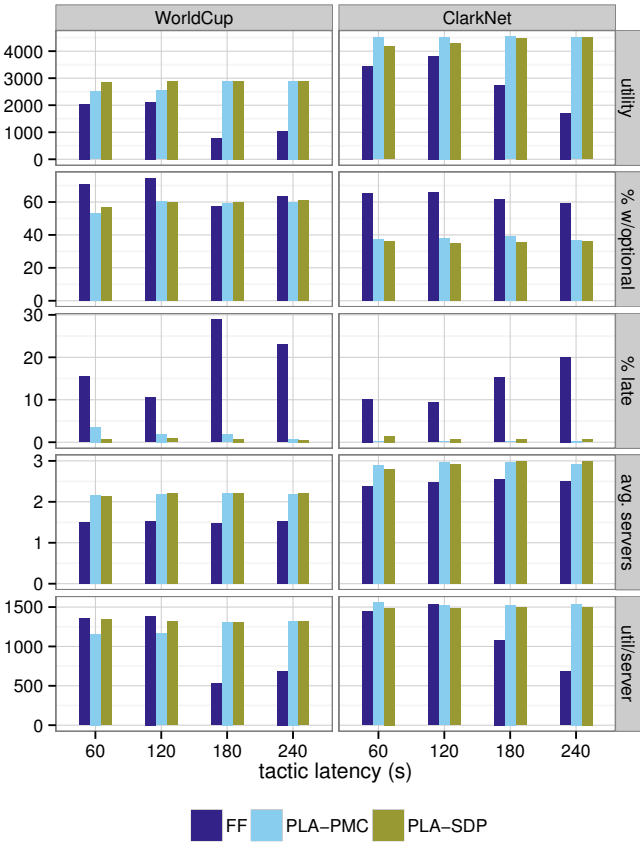

Fig. 7. Comparison of approaches

benefits of proactivity and latency-awareness using a different utility function, and an additional trace compared to the evaluation in our previous work. For each approach we ran the system four times, each with a different latency for the tactic to add a server ($\lambda = 60$, 120, 180, and 240 seconds). For each run, the first 15 minutes were used to let the system warm up (e.g., prime the time series predictor) with no adaptation, and self-adaptation was used during the remaining 1.5 hours of the run, during which the metrics for the evaluation were collected. The results of the comparison of the PLA approaches with FF are shown in Fig. 7. It can be observed that with the FF approach, the utility provided by the system drops as the tactic latency gets larger, whereas the PLA approaches are able to maintain the level of utility despite the increased latency. Additionally, we show other metrics that, even though are not the main criteria for adaptation, are interesting to observe. The FF approach provides more responses with optional content. This is understandable because a latency-agnostic approach ignores the fact that the tactic to change the dimmer is much faster than the tactic to add a server, thus favoring the latter, expecting to get a higher reward. However, the percentage of responses that do not meet the target response time increases with latency when latency is ignored, resulting in penalties instead. The PLA approaches, on the other hand, are able to keep the percentage of late responses very low

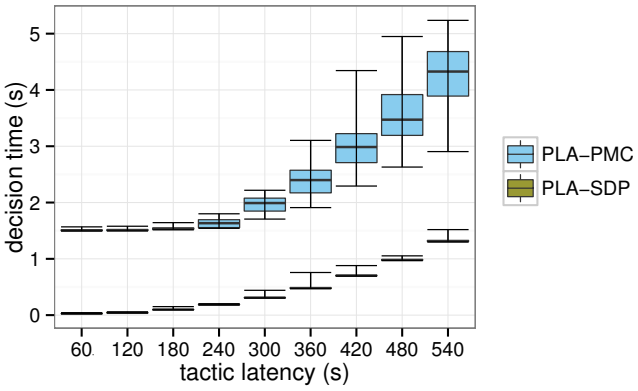

Fig. 8. Adapation decision times

in spite of the increase in tactic latency. The charts plotting the average utility per server show that despite using more servers, the PLA approaches obtain more utility per server. Also, we can observe that in these experiment runs, PLA-SDP produces results very similar to those of PLA-PMC. The slight difference between the two is due to uncontrolled disturbances in the runs, such as network delays, and background processes. In fact, we also ran the experiments using a simulation of the system that allows us to replicate exactly the conditions for both approaches [24], and confirmed that PLA-SDP makes exactly the same adaptation decisions as PLA-PMC.

To compare the running time of the run-time decision of PLA-SDP with PLA-PMC, we ran the experiments in simulation[8] with different values for the tactic latency λ. Larger values of λ result in longer look-ahead horizons, which in turn increase the state space that must be explored to find the solution to the adaptation decision problem. The results are shown in the box plot in Fig. 8, with the median, 1st and 3rd quartiles in the box, and the range in the bar. These results show that the adaptation decisions with PLA-SDP are faster than with PLA-PMC, with an average speedup of 16.2, and with much less variance. In summary, PLA-SDP makes adaptation decisions close to an order of magnitude faster than PLA-PMC while still producing the same results.

VI. Related Work

In previous work, we also used Alloy to compute reachability predicates for making PLA adaptation decisions using dynamic programming [24]. However, that work neither supported decisions for concurrent tactics, nor took into account the uncertainty of environment predictions. In PLA-PMC, we were able to make adaptation decisions supporting concurrent tactic execution by modeling tactics as parallel processes [7]. The probabilistic model checker naturally handles models with stochastic behavior, allowing the approach to consider the uncertainty of environment predictions. The approach presented in this paper maintains the features of PLA-PMC, while being able make adaptation decisions much faster. One limitation of PLA-PMC is that it requires implementing the computation of the utility function and its underlying model (LPS queueing equations in our example) in the PRISM language. PLA-SDP does not have this limitation, and allows invoking third-party tools such as layered queueing network solvers to compute the utility function.

There are approaches that use reinforcement learning to gradually learn the optimal policy for the underlying MDP [25], [26]. Their advantage is not requiring the construction of the MDP. However, they need time to learn the dynamics of the system, and have to execute possibly inadequate adaptations to learn their effect. Naskos et al. use MDPs to make cloud elasticity decisions [27]. Their approach focuses on tactics to add and remove servers, and consequently, it cannot decide between alternative tactics, nor supports concurrent tactics. In addition, their work uses the PRISM model checker at run time, as we do in PLA-PMC, thus having the run-time overhead that this paper addresses.

Our approach shares the high-level ideas of model predictive control (MPC), namely, (i) the use of a model to predict the future behavior of the system; (ii) the computation of a sequence of control actions, committing only to the first one; and (iii) the use of receding horizon [28]. Although MPC has been used in other approaches to self-adaptation [29], [30], to the best of our knowledge our approach differs in the following ways. First, it takes into account that control actions executed at a given time may prevent other control actions from being applicable in subsequent time steps, as opposed to assuming that all control actions are applicable at all times. Second, it considers tactic latency during the selection of the adaptation action(s), not just as an adaptation cost, but modeling how the execution of the tactics affects the applicability of other tactics while the tactics execute (over possibly multiple time intervals). Furthermore, our approach is able to decide between fast and slow adaptation tactics. Third, it considers the possible concurrent adaptation tactics during the decision, not just as a way to speed up the execution of the adaptation. Fourth, it considers the transition probabilities of the environment instead of treating the predictions for the environment state at each time interval over the decision horizon independently.

VII. Conclusion

We have presented an approach for proactive latency-aware adaptation that makes adaptation decisions faster while producing the same results as an approach based on probabilistic model checking. This can achieved by keeping the system and environment components of the MDP used to solve the adaptation decision problem separate as much as possible. The system MDP is difficult to build due to the possible combinations of tactics, system states, and the (in)compatibility of certain tactics. However, because of this separation, the system MDP does not require information about the environment, which is only known at run time. Therefore, it can be built offline using formal specification in Alloy. The probabilistic model of the environment is updated at run time, and is combined with

[8]The adaptation decision code is exactly the same used with the real system; only the managed system is simulated.

the system MDP as the adaptation decision is solved using stochastic dynamic programming. Our experimental results show that this approach is close to an order of magnitude faster than using probabilistic model checking at run time to make adaptation decisions, while preserving the same effectiveness advantage over an approach that is not latency aware.

Even though in this paper we used an multi-tier web system as an example, the approach is applicable to MAPE-K systems in general. In fact, this approach has been used for adaptation decisions in a team of unmanned aerial vehicles [31]. In both cases, the formal specification was handwritten. However, since it follows regular patterns, the specification could be generated automatically. In future work, we plan on generating these specifications from simpler specifications written in a tactic specification language like Stitch [32].

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. [Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. (DM-0003198).

REFERENCES

[1] B. H. Cheng *et al.*, "Software engineering for self-adaptive systems: A research roadmap," in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, B. H. C. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer Berlin Heidelberg, Jun. 2009, vol. 5525.

[2] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 2, May 2009.

[3] E. Yuan, N. Esfahani, and S. Malek, "A systematic survey of self-protecting software systems," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 8, no. 4, Jan. 2014.

[4] 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.

[5] 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*. IEEE, Jun. 2012.

[6] 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*. ACM, Jun. 2014.

[7] 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 - ESEC/FSE 2015*. ACM Press, Aug. 2015.

[8] "RUBiS: Rice University Bidding System," http://rubis.ow2.org/.

[9] 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*. IEEE, Jun. 2014.

[10] M. Islam, S. Ren, H. Mahmud, and G. Quan, "Online energy budgeting for cost minimization in virtualized data center," *IEEE Transactions on Services Computing*, vol. PP, no. 99, 2015.

[11] 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)*. IEEE, Jul. 2014.

[12] 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*. IEEE, Oct. 2014.

[13] 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 - ICSE 2014*. ACM, May 2014.

[14] G. Jung, K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and C. Pu, "A cost-sensitive adaptation engine for server consolidation of multitier applications," *Middleware 2009*, 2009.

[15] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, 2003.

[16] M. Puterman, "Dynamic programming," *Encyclopedia of Physical Science and Technology*, vol. 4, 2002.

[17] J. Zhang and B. Zwart, "Steady state approximations of limited processor sharing queues in heavy traffic," *Queueing Systems*, vol. 60, no. 3-4, Nov. 2008.

[18] D. Jackson, *Software Abstractions: logic, language, and analysis*. The MIT Press, 2012.

[19] C. A. R. Hoare, "Programs are predicates," in *Mathematical Logic and Programming Languages*, C. A. R. Hoare and J. C. Shepherdson, Eds. Prentice-Hall, 1985.

[20] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: verification of probabilistic real-time systems," in *23rd international conference on Computer Aided Verification*. Springer-Verlag, Jul. 2011.

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

[22] M. Arlitt and T. Jin, "A workload characterization study of the 1998 World Cup web site," *IEEE Network*, vol. 14, no. 3, 2000.

[23] M. F. Arlitt and C. L. Williamson, "Web server workload characterization," in *Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems - SIGMETRICS '96*, vol. 24, no. 1. ACM Press, May 1996.

[24] J. Cámara, G. A. Moreno, D. Garlan, and B. Schmerl, "Analyzing latency-aware self-adaptation using stochastic games and simulations," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 10, no. 4, Jan. 2016.

[25] M. Amoui, M. Salehie, S. Mirarab, and L. Tahvildari, "Adaptive action selection in autonomic software using reinforcement learning," in *Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08)*. IEEE, Mar. 2008.

[26] E. Barrett, E. Howley, and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, Aug. 2013.

[27] A. Naskos, E. Stachtiari, A. Gounaris, P. Katsaros, D. Tsoumakos, I. Konstantinou, and S. Sioutas, "Dependable horizontal scaling based on probabilistic model checking," in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, May 2015.

[28] E. F. Camacho and C. B. Alba, *Model Predictive Control*. Springer, 2013.

[29] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang, "Power and performance management of virtualized computing environments via lookahead control," *Cluster Computing*, vol. 12, no. 1, Oct. 2008.

[30] B. Trushkowsky, P. Bodík, A. Fox, and M. Franklin, "The SCADS director: Scaling a distributed storage system under stringent performance requirements." *FAST*, 2011.

[31] S. A. Hissam, S. Chaki, and G. A. Moreno, "High assurance for distributed cyber physical systems," in *Proceedings of the 2015 European Conference on Software Architecture Workshops*. ACM Press, Sep. 2015.

[32] S.-W. Cheng and D. Garlan, "Stitch: A language for architecture-based self-adaptation," *Journal of Systems and Software*, vol. 85, no. 12, Dec. 2012.