Flexible and Efficient Decision-Making for Proactive Latency-Aware Self-Adaptation

GABRIEL A. MORENO, Software Engineering Institute and Carnegie Mellon University
JAVIER CÁMARA, DAVID GARLAN, and BRADLEY SCHMERL, Carnegie Mellon University

Proactive latency-aware adaptation is an approach for self-adaptive systems that considers both the current and anticipated adaptation needs when making adaptation decisions, taking into account the latency of the available adaptation tactics. Since this is a problem of selecting adaptation actions in the context of the probabilistic behavior of the environment, Markov decision processes (MDPs) 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 has been used to deal with this problem, but it requires constructing the MDP every time an adaptation decision is made to incorporate the latest predictions of the environment behavior. In this article, we describe PLA-SDP, an approach that eliminates that runtime overhead by constructing most of the MDP offline. At runtime, the adaptation decision is made by solving the MDP through stochastic dynamic programming, weaving in the environment model as the solution is computed. We also present extensions that support different notions of utility, such as maximizing reward gain subject to the satisfaction of a probabilistic constraint, making PLA-SDP applicable to systems with different kinds of adaptation goals.

CCS Concepts: • **Computing methodologies → Markov decision processes**; *Planning under uncertainty*;

Additional Key Words and Phrases: Self-adaptive systems, proactive latency awareness, Markov decision process

ACM Reference format:
Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2018. Flexible and Efficient Decision-Making for Proactive Latency-Aware Self-Adaptation. *ACM Trans. Auton. Adapt. Syst.* 13, 1, Article 3 (April 2018), 36 pages.
https://doi.org/10.1145/3149180

This material is based upon work funded and supported by awards CNS 1116848 from the National Science Foundation, N000141310401 and N000141310171 from the Office of Naval Research, by the National Security Agency, and 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. Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. DM-0004522.
Authors' addresses: G. A. Moreno (corresponding author), Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 4500 Fifth Ave., PA 15213; email: gmoreno@sei.cmu.edu; J. Cámara, D. Garlan, and B. Schmerl, School of Computer Science, Carnegie Mellon University, Pittsburgh, 5000 Forbes Ave., PA 15213; emails: {jcmoreno, garlan, schmerl}@cs.cmu.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 ACM 1556-4665/2018/04-ART3 $15.00
https://doi.org/10.1145/3149180

1 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 [10, 37, 41]. Most self-adaptation approaches are reactive, making adaptation decisions based on current conditions [25]. 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 [29]. 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 2 minutes, it is not possible to reach a system configuration with one more server in 1 minute. The only way to have that additional server on time is to start its addition ahead of time (i.e., *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 that simply changes the QoS setting. 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 (PLA) 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 [8, 30]. 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 (MDPs) 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, both computationally and for a developer to do it without adequate abstractions.

In previous work, we proposed a proactive latency-aware approach to self-adaptation that uses probabilistic model checking to deal with this problem [30]. 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. From the solution to the MDP, we can extract 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 and the environment states in the MDP depend on the stochastic behavior of the environment, which can only be estimated at runtime, and with a short horizon. Consequently, the 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.

To address this issue, in subsequent work we presented PLA-SDP,[1] an approach that practically eliminates the runtime overhead of constructing the MDP by doing most of it *offline* [31]. This is possible because the MDP is separated into a system MDP, whose transitions are independent of the environment and thus can be constructed offline, and an environment model. To generate the system MDP, the approach uses formal modeling and analysis, exhaustively considering the many possible system states, and combinations of tactics, including their concurrent execution when they do not interfere with each other. At runtime, the adaptation decision is made by solving the MDP through stochastic dynamic programming, weaving in the stochastic environment model as the solution is computed.

This article extends the work described in [31] with support for a variety of forms of utility to drive adaptation decisions. In addition to the often used form of additive utility maximization, this extension allows combining different ways in which utility is gained with requirements on the satisfaction of a probabilistic constraint. This makes it possible to express, for example, that the system gains utility as long as it does not fail, and that the probability of failure must be kept above some threshold. This support for a variety of forms for the utility function gives more flexibility to PLA-SDP, making it applicable to systems with different types of adaptation goals.

To explain and evaluate the approach, we use RUBiS, a web application that has the core functionality of an auctions website [36]. Our experimental results show that this approach reduces the adaptation decision time by an order of magnitude compared to the probabilistic model checking approach described in [30], while still producing the same results. In this article, we also extend the evaluation of the approach with another system that is different from RUBiS in several ways, including their domains, their tactic repertoires, and their adaptation goals. This cyber-physical system called DART (Distributed Adaptive Real-Time) consists of a team of unmanned air vehicles flying a reconnaissance mission in a hostile environment. The adaptation goal presents a tension between detecting targets on the ground and keeping the probability of surviving the mission above a threshold. Such adaptation goals could have not been handled without the extensions presented in this article.

In addition to the new support for different forms of utility functions and the additional system used for evaluation, this article provides more details about several components of the approach, including an example of an environment model and how it is generated, and an explanation of how multiple tactics with latency are handled.

The rest of the article is organized as follows. RUBiS, the main example that is used throughout the article, is presented in Section 2. In Section 3, we describe the general adaptation model within which our adaptation decision approach works. The core of the approach is presented in Section 4. The extensions to support different notions of utility are presented in Section 5. The evaluation of the approach is described in Section 6, including the results as well as the description of DART, the additional system on which the approach is evaluated. Related work is presented in Section 7. Our conclusions and future work directions are in Section 8.

[1]PLA-SDP stands for *Proactive Latency-Aware with Stochastic Dynamic Programming*.

2 EXAMPLE

To illustrate the approach, we use RUBiS, an open source benchmark application that implements the functionality of an auctions website [36]. This application is widely used for research in web application performance, and various areas of cloud computing [12, 13, 19, 35]. 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 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.[2] 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.[3] The other pair of tactics leverages the *brownout* paradigm [24]. 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 [21]. 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

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

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 $R_O > R_M$.

3 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 [23]. The MAPE phases cover the activities

[2]The latency is assumed to be constant, but if it was a random variable, λ would be its expected value.
[3]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.

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.

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 a 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.

4 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 [30]. 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 constrains subsequent decisions, the adaptation decision problem is a sequential decision problem [34]. 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.

Figure 1 shows an overview of the elements of PLA-SDP and how they are used for making adaptation decisions. The strategy to reduce the time it takes to make an adaptation decision at runtime is to avoid the runtime overhead of constructing the MDP. Since the model of the environment needed to build the complete MDP is not known until runtime, constructing the complete MDP offline is not possible. However, we can separate the aspects of the MDP that are known before runtime—namely, the system model and how adaptation tactics affect the system state—from the environment model. In that way, the system MDP can be constructed offline. The system MDP is encoded as reachability predicates that specify whether a system configuration can be reached from another system configuration either through the passage of time, or with the use of one or more adaptation tactics. These predicates are formally defined in Section 4.1. In order to make runtime adaptation decisions, it is necessary to compute the list of system configuration pairs that

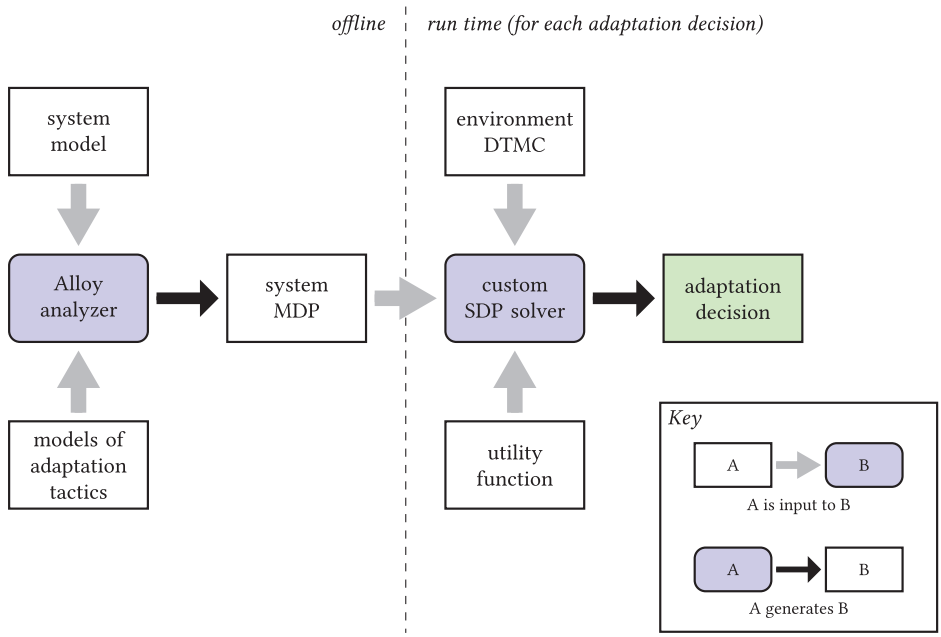


Fig. 1. Elements of PLA-SDP.

satisfy these predicates for that particular system with its adaptation tactics. This explicit representation of the predicates—and consequently of the system MDP—is computed from specifications of the system and its adaptation tactics using the Alloy analyzer [20], a tool that analyzes these specifications exhaustively to find all the possible configuration pairs that satisfy these predicates. This computation, described in Section 4.2, is done offline and results in the system MDP. When an adaptation decision has to be made at runtime, a stochastic model of the predicted evolution of the environment over the decision horizon is generated and encoded as a discrete-time Markov chain (DTMC), as described in Section 4.1.1. To make the decision, an algorithm based on the principles of stochastic dynamic programming computes the solution to the adaptation decision problem, weaving the environment DTMC into the system MDP as the solution is computed.

4.1 Adaptation Decision Problem Formulation

The formulation of the problem, and consequently its solution, are based on the following assumptions:

(1) The adaptation goal can be expressed with one of the utility forms presented in Section 5.
(2) Utility can be computed as a function of the state of the system and its environment. This, in turn, requires being able to estimate the system measures of performance, such as response time, that are used to compute utility.
(3) Adaptation tactics have a deterministic effect on the structure and properties of the system.
(4) The actions of the system through adaptation tactics do not affect the evolution of the environment.

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

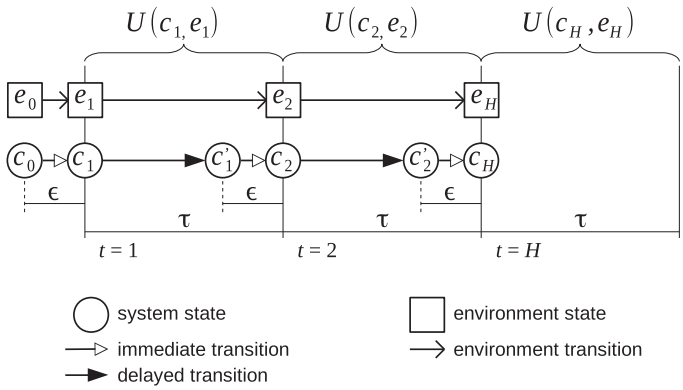


Fig. 2. Pattern of adaptation transitions in adaptation decision solution.

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.

Any solution to the adaptation decision problem will follow the pattern of immediate and delayed transitions shown in Figure 2 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.[4] 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 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

[4]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).

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 as follows:

- $R^I(c, c')$, 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 explained in Section 4.2), reducing the burden on the runtime decision algorithm.

These predicates define the transition matrix for the system portion of the adaptation MDP. Therefore, a solution like the one shown in Figure 2 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:

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

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

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

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

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

where $v^H(c)$ represents the value that each configuration has at the end of the horizon—the base cases—and $v^t(c)$ is the value that will be obtained from timestep t onwards if the system is in configuration c at time t and optimal decisions are made in the remainder of the horizon. The configuration that the system should adapt to in order to maximize the utility accumulated over the decision horizon is the one that has the maximum value at time $t = 1$.

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

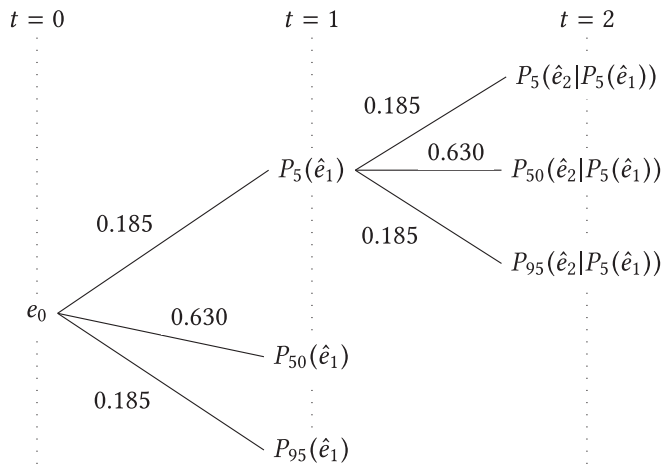


Fig. 3. Partial probability tree.

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 [43]. 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).[5]

The result of arg max in Equation (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 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.

4.1.1 Stochastic Environment. We model the evolution of the environment over the decision horizon as a DTMC. The set of environment states is denoted by E, and the probability of transitioning from state e to state e' is given by $p(e'|e)$. The set E and the transition probabilities for the environment can be obtained in different ways, and the adaptation decision algorithm is oblivious to how that is done.

For RUBiS, we used a time series predictor to predict future request rates based on past observations of the environment.[6] These predictions are used to generate a probability tree like the one partially shown in Figure 3. The root of the tree represents the state of the environment at the time the adaptation decision is being made ($t = 0$). All the nodes at tree depth i represent possible realizations of the environment in the period $t = i$. Even though there are infinite possible

[5]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.

[6]In Section 6.2.2, we present another example of environment model construction, which is based on sensing the physical environment in front of a cyber-physical system.

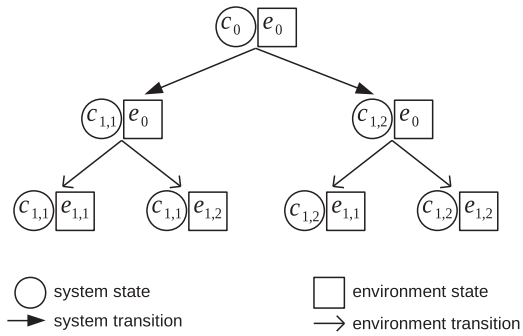


Fig. 4. System and environment transitions.

environment realizations, the probability distribution over environment realizations is discretized, and only a few possible environment values are considered. The children of each node represent realizations conditioned on the parent, with the edges representing the probability of the child realization given that the parent was realized.

The probability tree is generated with the technique described in our previous work [30], which uses the Extended Pearson-Tukey (EP-T) three-point approximation [22] to discretize the probability distribution of the estimation for the following period into three points. These points correspond to the 5th, 50th, and 95th percentiles of the estimation distribution, with probabilities 0.185, 0.630, and 0.185, respectively.

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,[7] 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 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 Figure 2, 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. Figure 4 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 [34], the adaptation decision problem with a stochastic model of the environment can be solved as follows:

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

[7] A DTMC can be turned into an MDP by assuming there is a single action applicable in every state.

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

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

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.

4.1.2 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 only necessary to keep track of the progress of the tactic 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 $p_{add} \in \{0,\ldots,\lceil \lambda/\tau \rceil\}$ is the number of time intervals left until the tactic completes, with 0 indicating that the tactic is not being executed.[8] 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 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.

4.2 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 modeling and analysis to compute the reachability predicates. Specifically, we use Alloy [20] 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, these

[8]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. 5. Alloy model: configurations.

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 runtime.

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 the serial application of any combination of compatible tactics would be the same as if they were applied in parallel.

To compute R^D, we use Alloy to find all the pairs $(c, c') \in \mathrm{CP} \times \mathrm{CP}$, such that $R^D(c, c')$, where CP is the configuration space extended with tactic progress. To achieve that, we first introduce other necessary pieces of the model. Figure 5 shows the declarations that define the system configuration space for our example.[9] 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 and 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 4.1.

Figure 6 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

[9]The complete Alloy models for RUBiS are included in Appendix A.

```
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. 6. Alloy model: configurations extended with tactic progress.

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 p from tactics with latency, LT, to the tactic progress, TP. The facts in lines 14 and 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.

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 Figure 7, 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 (line 3). If it reaches the last level of progress, then the configuration has one more server in the post-state (lines 5), 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 10), and does not have an effect (line 11). We also need to require that nothing else changes (lines 15 and 16). Line 15 requires that every field of C other than s stays the same, whereas line 16 ensures that the progress of all the other tactics has not been changed in the post-state.

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 (lines 19–21). If we had more than one tactic with latency, their predicates would have to be composed to reflect the effect that 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 [17]. For example, if we

```
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] // tactic effect
        } else {
            c'.s = c.s // no finished yet, maintain state
        }
    } else { // tactic is not running
        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. 7. Alloy model: tactic progress predicate.

had another tactic with latency for rebooting a server, the progress predicates for the tactics would be composed as follows:

```
some t : CP | addServerTacticProgress[c, t] and
rebootTacticProgress[t, c'].
```

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 runtime 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 it is applicable. 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. Even though this trace consists of a sequence of tactics, it represents the simultaneous start of those tactics, whose combined effect on the system state is the result of their sequential composition. Using the Alloy analyzer we can find all possible traces that satisfy this model, and the set of all pairs formed by the first and last state of each trace is the relation R^I.

Figure 8 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 Equation (7) is found. To accomplish that,

```
open util/ordering[TraceElement] as Trace

sig TraceElement {
    cp : CP,
    starts : set T // tactics started
}

fact traces {
    let fst = Trace/first | fst.starts = none
    all e : TraceElement - last | let e' = next[e] | {
            equals[e, e'] and equals[e', Trace/last]
        } or addServerTacticStart[e, e'] or removeServerTactic[e, e'] or
             decDimmerTactic[e, e'] or incDimmerTactic[e, e']
}
```

Fig. 8. Alloy model: traces for immediate reachability.

```
1  pred addServerCompatible[e : TraceElement] {
2      e.cp.p[AddServer] = TPASO/last
3      !(RemoveServer in e.starts)
4  }
5
6  pred addServerTacticStart[e, e' : TraceElement] {
7      addServerCompatible[e]
8      e.cp.s != SO/last
9
10     e'.starts = e.starts + AddServer
11     let c = e.cp, c'=e'.cp | {
12         c'.p[AddServer] = TPASO/first
13
14         // nothing else changes
15         equals[c, c']
16         (LT - AddServer) <: c.p in c'.p
17     }
18 }
```

Fig. 9. Alloy model: predicates for tactic start.

the elements of the trace have not only the configuration state, but also a set of tactics that have been started to arrive at that particular state in the trace (lines 3–6). The fact `traces` defines what a valid trace is. Line 9 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 the same as its predecessor, or is related to its predecessor by one of the predicates for the tactics. These predicates specify tactic applicability and how the state is affected when the tactic is started, which largely depends on whether the tactic has latency or not.

For each tactic with latency there is a predicate that models the start (but not the effect, which is delayed) of the tactic. An example is shown in Figure 9. These predicates relate trace elements instead of configuration states, since they have to maintain the state of `starts` in the trace elements, and use it to determine tactic compatibility. The predicate `addServerTacticStart` first checks that the tactic is compatible (line 7) using the predicate `addServerCompatible`. This predicate (lines 1–4) checks that this tactic has not already been started in the trace, and that it is

compatible. In this case, this tactic is not compatible with the tactic `RemoveServer`, so it ensures that the latter has not been started. Also, the predicate holds only if the tactic is applicable, which in this case means that the number of servers has not reached its maximum, or last value (line 8). In the post-state, the tactic is added to the set of tactics started (line 10), and the progress of the tactic is set to the first level (line 12). In addition, the predicate ensures that nothing else changes (lines 15 and 16).

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.

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 runtime. This map can be used to determine the tactics that have to be started to realize the immediate transition from the current configuration to the next optimal configuration found with Equation (7).

5 SUPPORT FOR ALTERNATIVE NOTIONS OF UTILITY

In previous sections, the adaptation goal has been presented as the maximization of utility accumulated over the execution of the system. Even though that characterization is sufficient to capture goals in many kinds of self-adaptive systems, there are others with different adaptation goals that cannot be coerced into the maximization of a sum. For example, suppose that reward (or utility) can only be gained as long as a constraint has always been satisfied since the system run started. In that case, whether the reward can be collected or not in an individual period depends on what has happened in previous periods. Even though this seems trivial as a rule for reward accounting, it is not that simple for making adaptation decisions. In PLA-SDP, the adaptation decision MDP is solved by backward induction, evaluating states in reverse chronological order. Therefore, either the algorithm requires a modification to deal with this, or the representation of state has to be expanded to include the necessary history.

In this section, we present an approach to support alternative notions of utility, allowing us to express adaptation goals as a combination of how reward is accumulated and a requirement on constraint satisfaction. The different combinations allow us to implement a variety of adaptation goals with an extension to PLA-SDP. One of the possible combinations is used for the case study described in Section 6.2, and other brief examples illustrating how other combinations can be used are provided in this section.

5.1 Adaptation Goal Composition

We define the adaptation goal as a reward maximization problem subject to keeping the probability of satisfying a constraint above a lower bound. The basic building blocks used to formulate the goal are the following functions, which determine the reward gain and constraint satisfaction during a single decision period with system configuration c in environment e:

— $g(c, e)$ is the reward that the system gains.
— $s(c, e)$ is the probability of satisfying a constraint.[10]

[10]For cases in which constraint satisfaction can be determined with certainty, the range of s can be $\{0, 1\}$. The approaches in this section can be used without modification.

Table 1. How Reward is Gained, Relative to the Constraint Satisfaction

When reward is gained	$\underset{c_1,\ldots,c_H}{\textbf{maximize}}$
RG1: *regardless of* the constraint satisfaction	$\sum_{t=1}^{H} g(c_t, e_t)$
RG2: *only when* the constraint is satisfied	$\sum_{t=1}^{H} s(c_t, e_t) g(c_t, e_t)$
RG3: *as long as* the constraint has been and is satisfied	$\sum_{t=1}^{H} \left(\prod_{i=1}^{t} s(c_i, e_i) \right) g(c_t, e_t)$

Table 2. Constraint Satisfaction Requirements (Zero or More)

Bound on the probability of	**subject to**
CS1: satisfying the constraint *in each period*	$\forall t \in \{1, \ldots, H\}\ s(c_t, e_t) \geq P$
CS2: satisfying the constraint *in every period*	$\prod_{t=1}^{H} s(c_t, e_t) \geq P$

For example, $g(c, e)$ could be the utility RUBiS attains by serving requests, and $s(c, e)$ could be the probability of the system being available (i.e., the probability of satisfying the constraint of not being down).

These functions refer to individual decision periods with no relation to what happens in other periods. However, the adaptation goal must be formulated over the whole decision horizon. Let us start with how reward is gained over the decision horizon, although that can itself be modified by the constraint. We define three reward gain (RG) forms relative to the satisfaction of the constraint, which are summarized in Table 1. RG1 is the reward accumulation over the decision horizon regardless of the constraint satisfaction, which is the adaptation goal we have used as an example in previous sections. In the other two forms, reward gain depends on the satisfaction of the constraint. In RG2, the reward is only gained in a given period if the constraint is satisfied in the period. Since $s(c, e)$ is a probability, RG2 maximizes the expected reward gained over the horizon. RG3 represents the case in which the reward is gained in a period as long as the constraint has been satisfied up to the current period (inclusive). The product in RG3 represents the probability of having satisfied the constraint in all the periods up to period t, and the whole equation is the expected reward gained over the decision horizon.

In addition to how reward is gained, it is also possible to impose a requirement on the probability of satisfying the constraint, although this is optional. Table 2 shows two different forms for the constraint satisfaction (CS) requirement. Form CS1 requires that the probability of satisfying the constraint be no less than a bound P in each period, whereas CS2 requires that the probability of satisfying the constraint in every period be no less than P.

Here are a few examples of how these reward gain and constraint satisfaction requirements can be used to formulate different adaptation goals.

Example 1. A system gets reward only if it is available, and it is desired to keep the probability of being available in each evaluation period over some threshold P. The fact that the system is unavailable in one period does not prevent it from being available afterward (i.e., it can repair itself). The problem formulation in this case is {RG2, CS1}, and the constraint is *being available*.

Example 2. A robot gets reward as long as it can operate (i.e., its battery has charge), and it is desired to keep the probability of being able to operate over some threshold P. If the robot runs out of battery, it cannot recover (e.g., it cannot reach a recharging station). The problem formulation in this case is {RG3, CS2}, with the constraint being *having battery charge*.

Example 3. A drone in a surveillance mission gets reward only if it is not detected in a segment of the route it is flying. However, being detected in a segment does not mean it cannot get reward in subsequent segments. Nevertheless, it is desired to keep the probability of being undetected during the entire mission over some threshold P. The problem formulation in this case is {RG2, CS2}, with the constraint being *not being detected*.

Example 4. A system gets reward in every period even if it is slow to respond. However, it is desired to keep the probability of not meeting the response time requirement in each period over some threshold P. The problem formulation in this case is {RG1, CS1}, with the constraint being *meeting the response time requirement*.

Example 5. A drone gets reward as long as it does not crash. The problem formulation in this case is {RG3}, with the constraint being *not crashing*.

5.2 PLA-SDP Formulation Extension

We now show how to extend the formulation of the stochastic dynamic programming adaptation decision to support these types of adaptation goals. In particular, we focus on the most difficult combination, {RG3, CS2}, since the other forms are either subsumed by this one or require minor modifications, as described later.

With an adaptation goal of the form {RG3, CS2}, the optimization problem is the following:

$$\underset{c_1,\ldots,c_H}{\text{maximize}} \quad \sum_{t=1}^{H}\left(\prod_{i=1}^{t} s(c_i,e_i)\right) g(c_t,e_t), \tag{8}$$

$$\text{subject to} \quad \prod_{t=1}^{H} s(c_t,e_t) \geq P. \tag{9}$$

The key idea to find a policy for the adaptation MDP that is a solution to this optimization problem is to avoid transitions in the policy that would result in a violation of constraint (9). However, achieving this requires keeping track of the probability of satisfying the constraint for each partial solution to the problem throughout the backward induction. In the following extended stochastic dynamic programming formulation of the adaptation decision problem, Equations (11) and (15) keep track of that probability, and Equations (12) and (18) filter out partial solutions that would violate Equation (9). In Equation (12), $C_t^T(c,e)$ is the set of configurations reachable in a time interval at time t from configuration c, when the environment state is e, such that constraint (9) is not violated. Analogously, C_0^I in Equation (18) is the set of configurations reachable immediately at time 0 from the current configuration, in the current environment state, such that constraint (9) is not violated.

$$v^H(c,e) = s(c,e)g(c,e), \quad \forall c \in C, e \in E_H, \tag{10}$$

$$S^H(c,e) = s(c,e), \quad \forall c \in C, e \in E_H. \tag{11}$$

For $t = H-1,\ldots,1$, and $\forall c \in C, e \in E_t$

$$C_t^T(c,e) = \left\{ c' \in C^T(c) \,\middle|\, s(c,e) \sum_{e' \in E_{t+1}} p(e'|e) S^{t+1}(c',e') \geq P \right\} \tag{12}$$

if $C_t^T(c,e) \neq \emptyset$

$$\hat{c}^t(c,e) \in \underset{c' \in C_t^T(c,e)}{\arg\max} \sum_{e' \in E_{t+1}} p(e'|e) v^{t+1}(c',e'), \tag{13}$$

$$v^t(c,e) = s(c,e)\left(g(c,e) + \sum_{e' \in E_{t+1}} p(e'|e) v^{t+1}(\hat{c}^t(c,e),e')\right), \tag{14}$$

$$S^t(c,e) = s(c,e) \sum_{e' \in E_{t+1}} p(e'|e) S^{t+1}(\hat{c}^t(c,e),e'); \tag{15}$$

otherwise,

$$v^t(c,e) = -\infty, \tag{16}$$

$$S^t(c,e) = 0. \tag{17}$$

Finally,

$$C_0^I = \left\{ c' \in C^I(c_0) \,\middle|\, \sum_{e' \in E_1} p(e'|e_0) S^1(c',e') \geq P \right\}, \tag{18}$$

$$C^* = \underset{c' \in C_0^I}{\arg\max} \sum_{e' \in E_1} p(e'|e_0) v^1(c',e'). \tag{19}$$

It is possible that $C_t^T(c,e) = \emptyset$ for some t, which means that there is no policy that satisfies the constraint satisfaction requirement. In that case, assuming that the system cannot stop, it is better to do the best possible. That requires solving the problem again, this time making C_t^T have the configuration that results in the highest probability of satisfying the constraint when no configuration meets that requirement. That is, if $C_t^T(c,e) = \emptyset$ in Equation (12), we can fall back to a best-effort approach and compute it as

$$C_t^T(c,e) = \underset{c' \in C^T(c)}{\arg\max}\, s(c,e) \sum_{e' \in E_{t+1}} p(e'|e) S^{t+1}(c',e'). \tag{20}$$

Analogously, if $C_0^I = \emptyset$ in Equation (18), we can compute a best-effort solution with

$$C_0^I = \underset{c' \in C^I(c_0)}{\arg\max} \sum_{e' \in E_1} p(e'|e_0) S^1(c',e'). \tag{21}$$

Note that it is not the same to directly compute the best-effort solution because it will enable paths that would be discarded by the first solution. So, if the first problem has a solution, it may not be the same as the best-effort solution.

The filtering done in Equation (12) filters out the target configurations that would violate the probability bound *in expectation* over the environment distribution. This filtering could have other forms. For example, the bound could be applied to the minimum probability of satisfaction, as follows.

$$C_t^T(c,e) = \{c' \in C^T(c) \mid s(c,e) \min_{e' \in E_{t+1}} S^{t+1}(c',e') \geq P\}. \tag{22}$$

The other forms of adaptation goals can be formulated based on the ones we have already presented. RG2 can be implemented as RG1 (which is the original formulation presented in Section 4)

by replacing $g(c, e)$ with $g'(c, e) = s(c, e)g(c, e)$. CS1 can be implemented as CS2, except that the tracking of the probability of satisfying the constraint in Equation (15) has to be replaced with

$$S^t(c, e) = s(c, e). \tag{23}$$

In addition to these extensions, there are other simple transformations that can be used to deal with other adaptation goals, and do not require changing the implementation of the decision algorithm. Changing the sign of $g(c, e)$ turns the goal into minimization. If instead of maximizing a sum as in RG1, the maximization of a product is required, that can be achieved by taking the logarithm of $g(c, e)$.

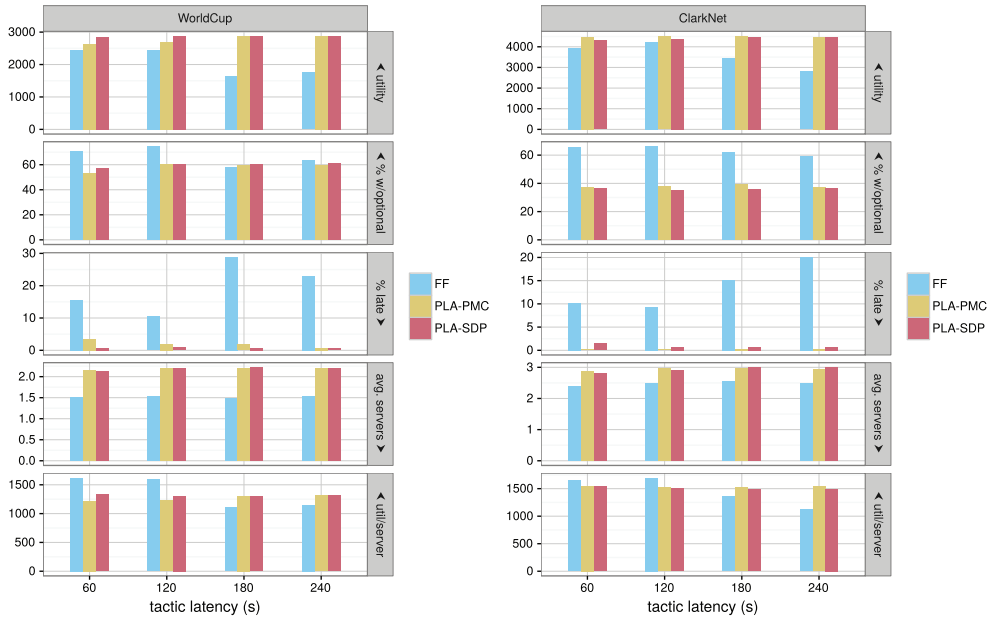
6 EVALUATION

We evaluate the approach using two systems that come from different domains, and have different adaptation goals and adaptation tactic repertoires. First, we use RUBiS to assess the adaptation decision speedup attained by PLA-SDP with respect to PLA-PMC, the approach based on probabilistic model checking [30]. PLA-PMC uses PRISM version 4.3 [27] to build and solve the MDP at runtime. In addition, the comparison with PLA-PMC is used to confirm that the effectiveness of the adaptation decision is not affected despite the speedup, maintaining the advantage of proactive latency-aware adaptation over latency-agnostic adaptation. To this end, we compare the adaptation effectiveness with a third adaptation approach that uses feed-forward (FF) adaptation and is latency agnostic. FF uses a one-step-ahead prediction of environment state to select the adaptation tactic that would result in the highest utility, assuming that tactics are instantaneous, and not looking beyond the current decision. The second system, DART, is used to compare the effectiveness of PLA-SDP over FF, leveraging the extensions that support other adaptation goals.

6.1 Evaluation with RUBiS

For the evaluation, RUBiS was deployed on a quad-core server running Ubuntu Server 14.04 as the host OS, with three virtual machines (VMs), 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 [24]. The load balancer HAProxy [15] 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 $H = \max(5, \lceil \lambda/\tau \rceil (S_{max} - 1) + 1)$, 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 five 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.



(a) WorldCup '98 trace. (b) ClarkNet trace.

Fig. 10. Comparison of approaches in RUBiS.

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 [3], and 1 day from the ClarkNet trace [4].[11] 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 benefits of proactivity and latency awareness. For each approach, we ran the system four times, each with a different latency for the tactic to add a server (λ = 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 Figure 10, with the arrows in the side labels pointing in the direction of better outcomes. 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 they 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

[11]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.

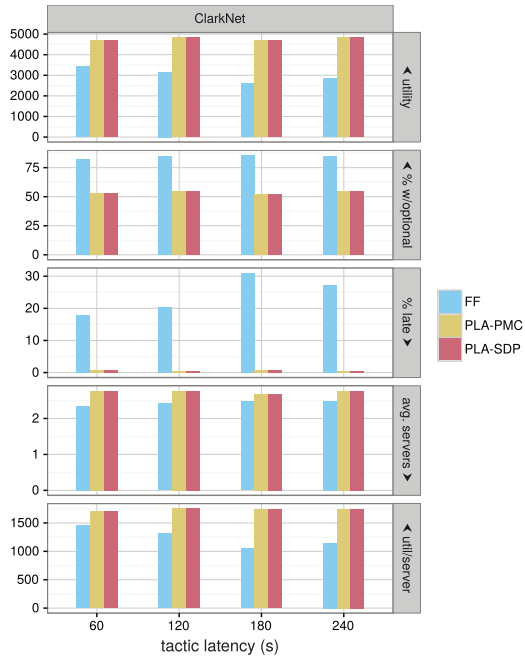
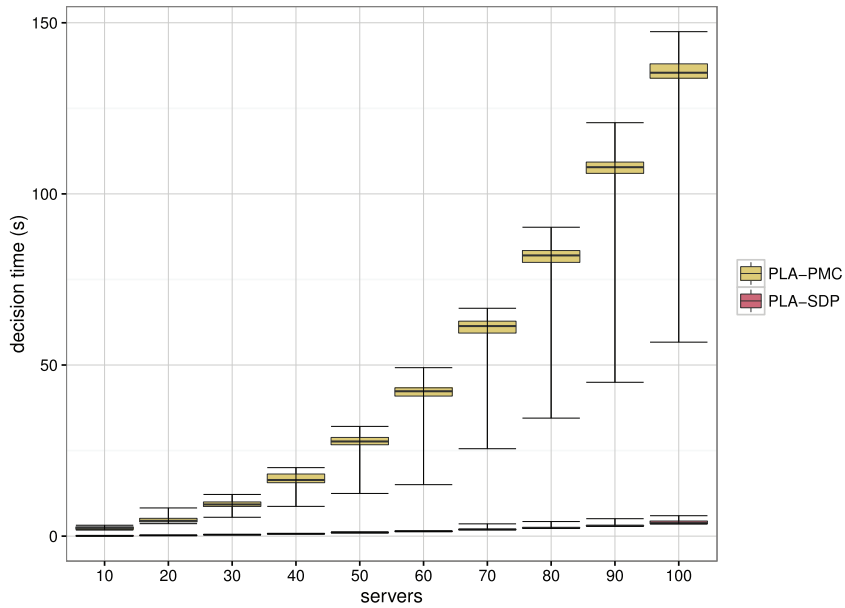


Fig. 11. Comparison of approaches in RUBiS with ClarkNet trace (simulation).

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 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 [9], and confirmed that PLA-SDP makes exactly the same adaptation decisions as PLA-PMC, as shown in Figure 11.

To compare the running time of the runtime decision of PLA-SDP with PLA-PMC, we measured the adaptation decision time with both solutions in a simulation of RUBiS.[12] Since the adaptation decision time increases with the size of the state space, the maximum number of servers used in RUBiS was varied between 10 and 100, resulting in increasing sizes of the state space. The results are shown in the box plot in Figure 12, with each box summarizing the statistics of the multiple decisions made in a run of the simulation, with the median, first and third quartiles represented by the box, and the range by the bar. These results show that the adaptation decisions with PLA-SDP are much faster than with PLA-PMC, with an average speedup of 27.9, and with much less variance. Despite computing the solution to the adaptation decision much faster, PLA-SDP produces exactly the same results as PLA-PMC.

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



Note. The boxes and range bars at the bottom correspond to PLA-SDP.

Fig. 12. Adaptation decision times with PLA-PMC and PLA-SDP.

6.2 Evaluation with DART

We first describe the DART system, its mission, and the formulation of its adaptation goal. Since, PLA-SDP relies on the construction of an environment model at runtime before each adaptation decision is made, we explain how that is accomplished for this system. We then present the evaluation results.

6.2.1 System Description. This system, developed in the context of the DART Systems project at the Carnegie Mellon® Software Engineering Institute [16], simulates a team of unmanned aerial vehicles (UAVs). The team flies in formation, with the designated leader drone at the center. High-level decisions, such as what formation to adopt, where to fly, or whether to move the formation up or down, are made autonomously by the leader (i.e., these UAVs are not remotely piloted). These decisions are then communicated to the rest of the team to be executed.

The simulated scenario embodies a tradeoff that a team of drones faces during a reconnaissance mission in a hostile environment. Namely, there is a tradeoff between detecting targets on the ground—the main purpose of the mission—and avoiding threats that could jeopardize the mission. The team's mission is to follow a planned route at constant forward speed, detecting as many targets on the ground as possible using a downward-looking sensor. Both targets and threats are static, but neither their number nor their location is known *a priori*. Therefore, self-adaptation is required for the team to execute the mission as best as it can, taking into account the information about the environment that is gathered during the mission, and the uncertainty that this information has.

The repertoire of adaptation tactics that the team can perform includes changing altitude (ascending or descending one altitude level), and adopting a particular formation (tight or loose).[13]

[13]Although these tactics change neither the architecture nor the configuration of the software system of DART, they do change the physical configuration of the system—the team of drones—by changing formation and the flying altitude.

The goal is to maximize the number of targets detected, taking into account that if the formation is lost to a threat, the mission fails. The lower the team flies, the more likely it is to detect targets, but also, the more likely it is to be hit by a threat. Changing formation also involves a similar tradeoff, since flying in tight formation reduces the probability of being hit by a threat, but at the same time reduces the chances of detecting targets.

The route is divided into D segments of equal length, and an adaptation decision is made periodically at the boundary between segments. Since the team flies at constant speed, we can refer to decision period t and segment t interchangeably. Let us define the configuration $c \in C$ of the team as the pair (a, ϕ), where a is the altitude and ϕ is the formation of the team. The state of the environment for segment i, denoted by e_i, is a pair (ρ_i, z_i), where ρ_i is the probability that it contains a target, and z_i is the probability that it contains a threat.

The expected number of targets detected during the mission can be computed as

$$q = \sum_{t=1}^{D} \left(\prod_{i=1}^{t} s(c_i, e_i) \right) g(c_t, e_t), \tag{24}$$

where $s(c_i, e_i)$ is the probability of survival at time i when the configuration of the team is c_i and the environment is e_i; and $g(c_t, e_t)$ is the probability of detecting a target at time t when the team is in configuration c_t and in environment e_t. The first factor in the summation in Equation (24) represents the probability of the team being operational at time t, which requires having survived since the start of the mission. The function s takes into account the probability of the presence of threats; that the probability of being destroyed is inversely proportional to the altitude; and how the formation affects that probability [40]. Similarly, function g considers the probability of the presence of targets; that the probability of detecting a target with the downward-looking sensor is inversely proportional to the altitude [38]; and that the formation also affects the detection probability [40].

If the self-adaptation goal were simply to maximize the expected number of targets detected during the whole mission according to Equation (24), it might happen that, upon not seeing any more targets ahead for the remainder of the mission, the team commits suicide, given that surviving will not help it detect more targets. One possible way to solve this issue is to include a reward for surviving the mission. However, the reward has to be calibrated with respect to the number of targets that could be detected during the mission. A reward too large would cause the team to be risk avoiding; but if it is too small, it will make the team take riskier behavior for a better chance at detecting targets. Given that the number of targets is not known *a priori*, this reward is difficult to calibrate. Furthermore, different missions may require different tradeoffs between target detection and survivability, and having to express the tradeoff by balancing rewards would be unwieldy.

Instead, we want to explicitly include the survivability requirement in the adaptation goal. Using one of the adaptation goal combinations introduced in Section 5—namely, {RG3, CS2}—we can express the DART adaptation problem over the decision horizon as

$$\begin{aligned} \underset{c_1, \ldots, c_H}{\text{maximize}} \quad & \sum_{t=1}^{H} \left(\prod_{i=1}^{t} s(c_i, e_i) \right) g(c_t, e_t), \\ \text{subject to} \quad & \prod_{t=1}^{H} s(c_t, e_t) \geq P. \end{aligned} \tag{25}$$

That is, targets can be detected as long as the survivability constraint is satisfied, and the probability of surviving must be at least P.

6.2.2 Environment Model. Before each adaptation decision, PLA-SDP takes as input a model of the stochastic behavior of the environment over the decision horizon. To gather the information needed to generate this environment at runtime, the team has a low-quality sensor[14] that makes observations of the segments ahead, giving an output of 1 for the detection of a threat in a segment, and 0 otherwise. The same is done for targets, and we assume that targets and threats can be sensed independently, through the use of two different sensors, or with a single sensor that can distinguish between both types of objects. In each monitoring interval, n samples are taken for each segment in the horizon of length H. The observations are accumulated over the monitoring intervals, so that when a segment first enters the look-ahead horizon, it will obtain n observations. In general, segment i in the horizon (with $i = 0$) for the current segment, will have $n(H - i + 1)$ samples. In order to build and maintain an environment model, it is necessary to keep track of two numbers for each segment and each object type: the number of detections, and the number of non-detections (or equivalently, the number of samples taken, and the number of detections).

To build the environment model, we assume that the two random variables in the environment state (i.e., probability of segment containing a threat, and a target, respectively) are independent.[15] Given that, we can construct two independent environment models, and then join them to produce the joint environment model. We first describe how the independent threat and target environment models are created, and then explain how they are joined.

Since the team visits one segment per decision interval, t timesteps into the future, the team will be t segments further down the route. Using the information captured by the environment monitoring, we can describe the probability density for ρ_t and z_t using the Beta distribution [6]. For each segment, the number of detections and non-detections correspond to the parameters α and β of the Beta distribution, respectively. This continuous distribution can then be discretized using the EP-T three-point approximation [22], allowing us to consider three possible realizations of the environment for each segment. We assume no dependencies between the state of different segments. Instead, we assume that a given state of the environment at time $t + 1$ can be reached with equal probability from every possible state at time t. Consequently, creating a tree to represent the evolution of the environment, as we did for RUBiS, would result in unnecessary replication of environment states. To avoid that, we represent the environment model as a DTMC with the topology exemplified in Figure 13 for a model of the threats with a horizon of length 3. The root corresponds to the state of the environment at the current time/segment. Since the environment model only covers the lookahead horizon for the adaptation decision, we refer to the current time as $t = 0$. For each value of $t \in \{1, \ldots, H\}$ there are three nodes corresponding to the three-point approximation of the distribution of the environment state at time t. Every node for $t < H$ has one edge going to each of the nodes that correspond to $t + 1$, with their probabilities set according to the EP-T discretization.

Once the independent environment models for the threats and the targets have been created as described above, the joint environment model is built by generating every possible combination of threat and target environment states, and creating the edges accordingly. Note that only combinations of states that correspond to the same time are feasible, and this fact can be exploited to reduce the state space. Therefore, the joint model will have a single root node, and for each $t \in \{1, \ldots, H\}$ it will have nine nodes.

[14]The sensor has false positive and false negative rates greater than zero.

[15]If they were not independent, their joint probabilities would have to be considered directly when building the joint environment DTMC. Since the resulting DTMC is an input to PLA-SDP, how the DTMC is constructed does not affect the decision algorithm.

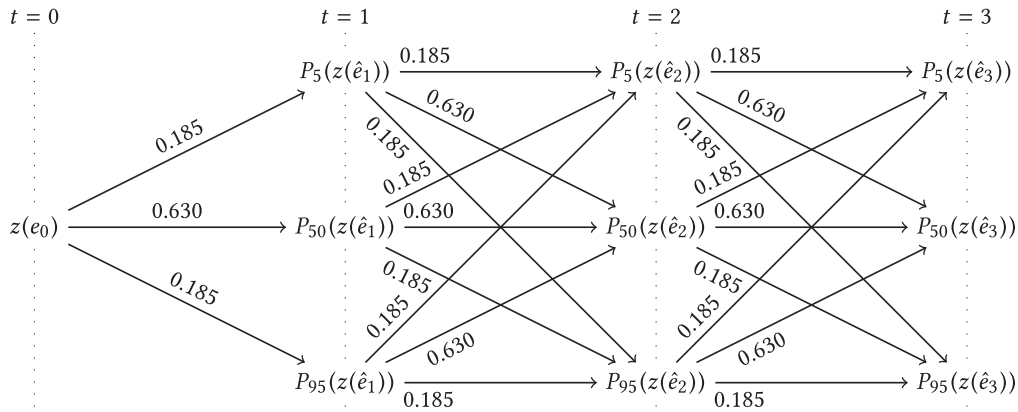


Fig. 13. DART environment model for threats ($H = 3$).

6.2.3 Evaluation Results with DART. Given the adaptation goal for this system, we used PLA-SDP with the extension for the {RG3, CS2} combination described in Section 5. For these experiments, the probability bound for the survivability constraint was set to $P = 0.90$. PLA-SDP was run with a horizon $H = 5$.

As a baseline for comparison and measurement of the effectiveness improvement, we also use a FF approach, which is latency agnostic, and not proactive, in that it does not look ahead further than the segment it is about to enter. More precisely, it uses a single-point estimation of the environment state in the segment that it is about to enter when making the adaptation decision. It is worth noting that the FF approach was implemented to also consider the survivability requirement in addition to the maximization of detected targets. This consideration, however, is limited to the period starting at the time the decision is being made, given the lack of look-ahead in FF.

The following results are based on 5,000 runs of the DART simulation with each approach. For each run, there were 20 targets and 7 threats randomly placed along a route of 100 segments. In addition, there are several other random behaviors in the simulation of the mission. The forward-looking sensors and the target sensor are subject to random effects, as is the impact of threats on the drones. Despite the large number of runs, for better comparison, the random number generators that control all of these random effects were seeded with matching seeds for the same run number of the two solution approaches. In that way, both approaches faced the same behavior of the environment.

The tactics used in this simulation control the altitude and the formation of the team. There are tactics to increase and decrease the altitude level at which the drones fly. The airspace is divided vertically into 10 altitude levels. Transitioning from one level to the next, up or down, takes the same time the team takes to traverse a segment in the route. Therefore, the latency of these tactics is τ; that is, the same as the decision interval. There are two other tactics that allow the team to change formation. One takes the team to a close formation, and the other to a tight formation. These two tactics are assumed to be immediate, or have negligible latency compared to the decision interval.

Figure 14 shows the proportion of missions in which the team survived with each approach. The team survived less than 10% of the missions with FF, while PLA-SDP satisfied the requirement of surviving with at least 90% probability.

Figure 15 shows the statistics for the number of targets detected with each approach. When considering all the missions, even those in which the team failed to survive, Figure 15(a) shows that

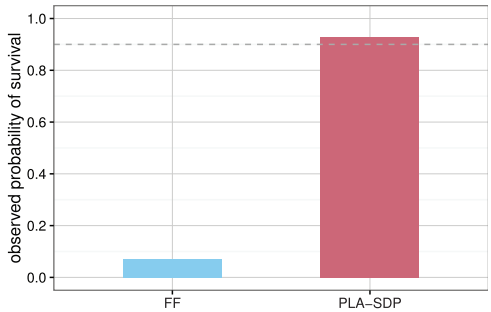
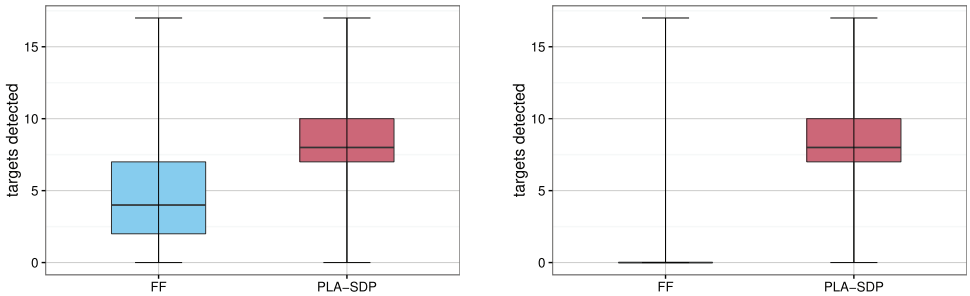


Fig. 14. Probability of mission survival in DART.



(a) Targets detected regardless of mission survival. (b) Targets detected adjusted for mission survival.

Fig. 15. Targets detected in DART.

PLA-SDP detected the highest number of targets on average, and in addition it has less variance. The average improvement of PLA-SDP over FF in the 5,000 missions simulated was 74%, with 3.5 more targets detected per mission on average. Depending on the drones and/or the mission, the team may not be able to transmit detected targets back to the base, requiring the team to complete the mission before the data can be downloaded. Adjusting the target count so that targets detected are counted only if the drones survive the mission, Figure 15(b) shows that the difference in effectiveness is even more pronounced. In this case, even the third quartile for FF is 0 because it only survived and detected targets in 7% of the missions.

7 RELATED WORK

Although there is related work in the area of self-adaptive systems that uses proactivity and considers adaptation latency, the PLA approaches presented in our previous work [30, 31] and in this article are different in that they consider adaptation latency systematically as a first-class concern.

Adaptation latency is considered in some very specific situations in some work [14, 18, 42]. For example, Gandhi et al. consider the setup time of servers for dynamic capacity management in data centers [14]. However, their work is specifically tailored to adding and removing servers to a dynamic pool. Even though that setting resembles the RUBiS example used in this article, their work cannot reason about other tactics that could be used instead of, or in combination with, tactics to control the number of servers. Zhang et al. propose a safe adaptation approach that can minimize the duration of the adaptation [42]. However, the duration of the adaptation is only considered to select one of all the possible ways of reaching the desired target configuration, and has no impact on the decision of what the target configuration should be. Iannucci and Abdelwahed consider the

latency of security actions when planning how to adapt to deal with security attacks [18]. In that case, latency is only considered in the decision by penalizing actions with longer latencies, but not actually taking into account how latency determines the time at which actions will change the state of the system. An important difference between these other works and ours is that PLA takes into account how the environment could change while the adaptation is carried out, what adaptation tactics would be (in)feasible during the execution of an adaptation, how utility changes while the system adapts, and how faster tactics can be used to complement slower tactics using concurrent execution.

In previous work, we also used Alloy to compute reachability predicates for making PLA adaptation decisions using dynamic programming [9]. 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 [30]. The probabilistic model checker naturally handles models with stochastic behavior, allowing the approach to consider the uncertainty of environment predictions. PLA-SDP 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 the RUBiS example) in the PRISM language. PLA-SDP does not have this limitation, and allows invoking third-party tools such as layered queueing network solvers [28, 33] to compute the utility function.

There are approaches that use reinforcement learning to gradually learn the optimal policy for the underlying MDP [1, 5]. 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 [32]. Their approach focuses on tactics to add and remove servers, and consequently, it cannot decide between alternative tactics, nor does it support concurrent tactics. In addition, their work uses the PRISM model checker at runtime, as we do in PLA-PMC, thus having the runtime overhead that this article addresses. Iannucci and Abdelwahed use MDPs to compute policies to deal with security attacks [18]. The main difference with our use of MDPs is that their work does not consider how the environment evolves over time while the system is adapting, focusing only on how the system state evolves. In addition, they only consider latency to favor faster tactics, since their approach is tailored to dealing with security attacks, in which it is desired to contain or clean the attack as fast as possible.

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 [7]. Although MPC has been used in other approaches to self-adaptation [2, 26, 39], 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 timesteps, 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.

8 CONCLUSION

We have presented PLA-SDP, 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 runtime. Therefore, it can be built offline using formal specification in Alloy. The probabilistic model of the environment is updated at runtime, and is combined with the system MDP as the adaptation decision is solved using stochastic dynamic programming. In addition, we presented extensions to PLA-SDP that support different notions of utility, formed by combining the way in which reward is gained relative to the satisfaction of a constraint, and a requirement on the satisfaction of that constraint. These extensions give more flexibility to PLA-SDP, making it applicable to different systems with different types of adaptation goals.

Our experimental results show that this approach is close to an order of magnitude faster than using probabilistic model checking at runtime to make adaptation decisions, while preserving the same effectiveness advantage over an approach that is not latency aware. In addition, we have shown that the effectiveness improvement is attained in two systems from different domains, with different adaptation goals and tactic repertoires.

For our experiments, the formal specifications were handwritten. However, these specifications have repetitive patterns for their different parts, such as tactics with latency and tactics without latency. Therefore, a large part of these specifications could be generated automatically. In future work, it would be possible to generate them from simpler specifications written in a tactic specification language like Stitch [11].

APPENDIX

A ALLOY MODELS FOR RUBIS

This appendix includes the complete Alloy models used to compute the reachability predicates for RUBiS.

A.1 Immediate Reachability Model

The following listing contains the PLA-SDP Alloy model for computing immediate reachability for RUBiS.

```
open util/ordering[S] as servers
open util/ordering[TAP] as progress // tactic add progress
open util/ordering[TraceElement] as trace
open util/ordering[D] as dimmer
open util/ordering[T] as TO

abstract sig TP {} // tactic progress
sig TAP extends TP {} // one sig for each tactic with latency

abstract sig T {} // tactics
abstract sig LT extends T {} // tactics with latency
one sig IncDimmer, DecDimmer, RemoveServer extends T {} // tactics with no latency
```

```
one sig AddServer extends LT {} // tactics with latency

// define configuration properties
sig S {} // the different number of active servers
sig D {} // the different dimmer levels

/* each element of C represents a configuration */
abstract sig C {
    s : S, // the number of active servers
    d : D // dimmer level
}

pred equals[c, c2 : C] {
    all f : C$.fields | c.(f.value) = c2.(f.value)
}

pred equalsExcept[c, c2 : C, ef : univ] {
    all f : C$.fields | f=ef or c.(f.value) = c2.(f.value)
}

/*
 * this sig is a config extended with the progress of each tactic with latency
 */
sig CP extends C {
    p: LT -> TP
} {
    ~p.p in iden // functional (i.e., p maps each tactic to at most one progress)
    //#p = #LT // every tactic in LT has a mapping in p
    p.univ = LT // every tactic in LT has a mapping in p (p.univ is domain(p) )
    p[AddServer] in TAP // restrict each tactic to its own progress class
}

fact tacticOrdering {
    TO/first = AddServer
    AddServer.next = RemoveServer
    RemoveServer.next = IncDimmer
    IncDimmer.next = DecDimmer
}

sig TraceElement {
    cp : CP,
    starts : set T // tactics started
}

// do not generate atoms that do not belong to the trace
fact {
    CP in TraceElement.cp
}

pred equals[e, e2 : TraceElement] {
    all f : TraceElement$.subfields | e.(f.value) = e2.(f.value)
```

```
}

fact traces {
    let fst = trace/first | fst.starts = none
    all e : TraceElement - last | let e' = next[e] | {
        equals[e, e']
        equals[e', trace/last]
    } or  ((addServerTacticStart[e, e'] or removeServerTactic[e, e'] or
         decDimmerTactic[e, e'] or incDimmerTactic[e, e']) and
        (let s = e'.starts - e.starts | all t : s | validOrder[t, e]))
}

pred validOrder[t : T, e : TraceElement] {
    all s : T | s in e.starts => !(s in t.nexts)
}

pred addServerCompatible[e : TraceElement] {
    e.cp.p[AddServer] = progress/last
    !(RemoveServer in e.starts)
}

pred addServerTacticStart[e, e' : TraceElement] {
    addServerCompatible[e] and e.cp.s != servers/last
    e'.starts = e.starts + AddServer
    let c = e.cp, c'=e'.cp | {
        c'.p[AddServer] = progress/first

        // nothing else changes
        equals[c, c']
        (LT - AddServer) <: c.p in c'.p
    }
}

pred removeServerCompatible[e : TraceElement] {
    !(RemoveServer in e.starts)
    e.cp.p[AddServer] = progress/last // add server tactic not running
}

pred removeServerTactic[e, e' : TraceElement] {
    removeServerCompatible[e] and e.cp.s != servers/first
    e'.starts = e.starts + RemoveServer
    let c = e.cp, c'=e'.cp | {
        c'.s = servers/prev[c.s]

        // nothing else changes
        equalsExcept[c, c', C$s]
        c'.p = c.p
    }
}

pred incDimmerCompatible[e : TraceElement] {
```

```
    !(IncDimmer in e.starts) and !(DecDimmer in e.starts)
}

pred incDimmerTactic[e, e' : TraceElement] {
    incDimmerCompatible[e] and e.cp.d != dimmer/last
    e'.starts = e.starts + IncDimmer

    let c = e.cp, c'=e'.cp | {
        c'.d = c.d.next

        // nothing else changes
        equalsExcept[c, c', C$d]
        c'.p = c.p
    }
}

pred decDimmerCompatible[e : TraceElement] {
    !(DecDimmer in e.starts) and !(IncDimmer in e.starts)
}

pred decDimmerTactic[e, e' : TraceElement] {
    decDimmerCompatible[e] and e.cp.d != dimmer/first
    e'.starts = e.starts + DecDimmer

    let c = e.cp, c'=e'.cp | {
        c'.d = c.d.prev

        // nothing else changes
        equalsExcept[c, c', C$d]
        c'.p = c.p
    }
}


pred show {
}

// the scope for TraceElement, C and CP has to be one more than the maximum
// number of tactics that could be started concurrently
run show for exactly 3 S, exactly 3 TAP, 2 D, 3 C, 3 CP, 3 TraceElement
```

A.2 Delayed Reachability Model

The following listing contains the PLA-SDP Alloy model for computing delayed reachability for RUBiS.

```
open util/ordering[S] as servers
open util/ordering[TAP] as progress // tactic add progress
open util/ordering[D] as dimmer

abstract sig TP {} // tactic progress
sig TAP extends TP {} // one sig for each tactic with latency
```

```
7
8  abstract sig T {} // tactics
9  abstract sig LT extends T {} // tactics with latency
10 one sig IncDimmer, DecDimmer, RemoveServer extends T {} // tactics with no latency
11 one sig AddServer extends LT {} // tactics with latency
12
13 // define configuration properties
14 sig S {} // the different number of active servers
15 sig D {} // the different dimmer levels
16
17 /* each element of C represents a configuration */
18 abstract sig C {
19     s : S, // the number of active servers
20     d : D // dimmer level
21 }
22
23 pred equals[c, c2 : C] {
24     all f : C$.fields | c.(f.value) = c2.(f.value)
25 }
26
27 pred equalsExcept[c, c2 : C, ef : univ] {
28     all f : C$.fields | f=ef or c.(f.value) = c2.(f.value)
29 }
30
31
32 fact uniqueInstances { all disj c, c2 : CP | !equals[c, c2] or c.p != c2.p}
33
34
35 /*
36  * this sig is a config extended with the progress of each tactic with latency
37  */
38 sig CP extends C {
39     p: LT -> TP
40 } {
41     ~p.p in iden // functional (i.e., p maps each tactic to at most one progress)
42     //#p = #LT // every tactic in LT has a mapping in p
43     p.univ = LT // every tactic in LT has a mapping in p (p.univ is domain(p) )
44     p[AddServer] in TAP // restrict each tactic to its own progress class
45 }
46
47
48 pred addServerTacticProgress[c, c' : CP] {
49     c.p[AddServer] != progress/last implies { // tactic is running
50         c'.p[AddServer] = progress/next[c.p[AddServer]]
51         c'.p[AddServer] = progress/last implies c'.s = servers/next[c.s] else c'.s = c.s
52     } else {
53         c'.p[AddServer] = progress/last // stay in not running state
54         c'.s = c.s
55     }
56
57     // nothing else changes other than s and the progress
```

```
    equalsExcept[c, c', C$s]
    (LT - AddServer) <: c.p in c'.p
}

pred oneStepProgress[c, c' : CP] { // is c' reachable from config c in one evaluation period?
    addServerTacticProgress[c, c']
}

sig Result {
    c, c' : CP
} {
    oneStepProgress[c, c']
}

// this reduces the number of unused configurations
fact reduceUsedConfigs {
    all cp : CP | {some r : Result | r.c = cp or r.c' = cp
        }
}

pred show {
}

/*
 * (numOfTacticsWithLatency + 1) for CP and C to allow the progress for all the tactics with latency + the initial state
 */
run show for exactly 3 S, exactly 3 TAP, exactly 2 D, 2 C, 2 CP, exactly 1 Result
```

REFERENCES

[1] Mehdi Amoui, Mazeiar Salehie, Siavash Mirarab, and Ladan Tahvildari. 2008. Adaptive action selection in autonomic software using reinforcement learning. In *Proceedings o f the 4th International Conference on Autonomic and Autonomous Systems (ICAS'08)*. IEEE, 175–181. DOI: http://dx.doi.org/10.1109/ICAS.2008.35

[2] Konstantinos Angelopoulos, Alessandro V. Papadopoulos, Vítor E. Silva Souza, and John Mylopoulos. 2016. Model predictive control for software systems with CobRA. In *Proceedings of the 11th International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'16)*. ACM, 35–46. DOI: http://dx.doi.org/10.1145/2897053.2897054

[3] Martin F. Arlitt and T. Jin. 2000. A workload characterization study of the 1998 World Cup web site. *IEEE Network* 14, 3 (2000), 30–37. DOI: http://dx.doi.org/10.1109/65.844498

[4] Martin F. Arlitt and Carey L. Williamson. 1996. Web server workload characterization. *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'96)* 24 (May 1996), 126–137. DOI: http://dx.doi.org/10.1145/233013.233034

[5] Enda Barrett, Enda Howley, and Jim Duggan. 2013. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience* 25, 12 (Aug. 2013), 1656–1674. DOI: http://dx.doi.org/10.1002/cpe.2864

[6] L. F. Bertuccelli and J. P. How. 2005. Robust UAV search for environments with imprecise probability maps. In *Proceedings of the 44th IEEE Conference on Decision and Control*. IEEE, 5680–5685. DOI: http://dx.doi.org/10.1109/CDC.2005.1583068

[7] Eduardo F. Camacho and Carlos Bordons Alba. 2013. *Model Predictive Control*. Springer. 405 pages.

[8] Javier Cámara, Gabriel A. Moreno, and David Garlan. 2014. 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'14)*. ACM, New York, 155–164. DOI: http://dx.doi.org/10.1145/2593929.2593933

[9] Javier Cámara, Gabriel A. Moreno, David Garlan, and Bradley Schmerl. 2016. Analyzing latency-aware self-adaptation using stochastic games and simulations. *ACM Transactions on Autonomous and Adaptive Systems* 10, 4 (Jan. 2016), 1–28. DOI: http://dx.doi.org/10.1145/2774222

[10] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. 2009. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, Betty H. C. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee (Eds.). Lecture Notes in Computer Science, Vol. 5525. Springer, Berlin, 1–26. DOI: http://dx.doi.org/10.1007/978-3-642-02161-9

[11] Shang-Wen Cheng and David Garlan. 2012. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software* 85, 12 (Dec. 2012), 2860–2875. DOI: http://dx.doi.org/10.1016/j.jss.2012.02.060

[12] Subhasri Duttagupta, Rupinder Virk, and Manoj Nambiar. 2014. Predicting performance in the presence of software and hardware resource bottlenecks. In *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'14)*. IEEE, 542–549. DOI: http://dx.doi.org/10.1109/SPECTS.2014.6879991

[13] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. 2014. Modeling the impact of workload on cloud resource scaling. In *Proceedings of the 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. IEEE, 310–317. DOI: http://dx.doi.org/10.1109/SBAC-PAD.2014.16

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

[15] HAProxy 2016. The Reliable, High Performance TCP/HTTP Load Balancer. Retrieved February 20, 2017 from http://www.haproxy.org/.

[16] Scott A. Hissam, Sagar Chaki, and Gabriel A. Moreno. 2015. High assurance for distributed cyber physical systems. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*. ACM, New York, 1–4. DOI: http://dx.doi.org/10.1145/2797433.2797439

[17] C. A. R. Hoare. 1985. Programs are predicates. In *Mathematical Logic and Programming Languages*, C. A. R. Hoare and J. C. Shepherdson (Eds.). Prentice-Hall, 141–154.

[18] Stefano Iannucci and Sherif Abdelwahed. 2016. A probabilistic approach to autonomic security management. In *Proceedings of the 2016 IEEE International Conference on Autonomic Computing (ICAC'16)*. IEEE, 157–166. DOI: http://dx.doi.org/10.1109/ICAC.2016.12

[19] Mohammad Islam, Shaolei Ren, Hasan Mahmud, and Gang Quan. 2015. Online energy budgeting for cost minimization in virtualized data center. *IEEE Transactions on Services Computing* PP, 99 (2015), 1–1. DOI: http://dx.doi.org/10.1109/TSC.2015.2390231

[20] Daniel Jackson. 2012. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

[21] Gueyoung Jung, Kaustubh R. Joshi, Matti A. Hiltunen, Richard D. Schlichting, and Calton Pu. 2009. A cost-sensitive adaptation engine for server consolidation of multitier applications. In *Proceedings of the 10th International Middleware Conference (Middleware'09), ACM/IFIP/USENIX*, Jean Bacon and Brian F. Cooper (Eds.). Springer, 163–183.

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

[23] Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50.

[24] Cristian Klein, Martina Maggio, Karl-Erik Arzén, and Francisco Hernández-Rodriguez. 2014. Brownout: Building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM, New York, 700–711. DOI: http://dx.doi.org/10.1145/2568225.2568227

[25] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. 2014. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing* 17, Part B (Oct. 2014), 184–206. DOI: http://dx.doi.org/10.1016/j.pmcj.2014.09.009

[26] Dara Kusic, Jeffrey O. Kephart, James E. Hanson, Nagarajan Kandasamy, and Guofei Jiang. 2008. Power and performance management of virtualized computing environments via lookahead control. *Cluster Computing* 12, 1 (Oct. 2008), 1–15. DOI: http://dx.doi.org/10.1007/s10586-008-0070-y

[27] Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of probabilistic real-time systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification*. Springer-Verlag, 585–591.

[28] LQNS 2011. Layered Queueing Network Solver. Retrieved February 20, 2017 from http://www.sce.carleton.ca/rads/lqns.

[29] Ming Mao and Marty Humphrey. 2012. A performance study on the VM startup time in the cloud. In *Proceedings of the 2012 IEEE 5th International Conference on Cloud Computing*. IEEE, 423–430. DOI: http://dx.doi.org/10.1109/CLOUD.2012.103

[30] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2015. 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'15)*. ACM Press, New York, 1–12. DOI: http://dx.doi.org/10.1145/2786805.2786853

[31] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2016. Efficient decision-making under uncertainty for proactive self-adaptation. In *Proceedings of the 2016 IEEE International Conference on Autonomic Computing (ICAC'16)*. IEEE, 147–156. DOI: http://dx.doi.org/10.1109/ICAC.2016.59

[32] Athanasios Naskos, Emmanouela Stachtiari, Anastasios Gounaris, Panagiotis Katsaros, Dimitrios Tsoumakos, Ioannis Konstantinou, and Spyros Sioutas. 2015. Dependable horizontal scaling based on probabilistic model checking. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 31–40. DOI: http://dx.doi.org/10.1109/CCGrid.2015.91

[33] OPERA n.d. Optimization, Performance Evaluation and Resource Allocator. Retrieved February 20, 2017 from http://www.ceraslabs.com/technologies/opera.

[34] M. L. Puterman. 2002. Dynamic programming. *Encyclopedia of Physical Science and Technology* 4 (2002), 673–696.

[35] Kashifuddin Qazi, Yang Li, and Andrew Sohn. 2014. Workload prediction of virtual machines for harnessing data center resources. In *Proceedings of the 2014 IEEE 7th International Conference on Cloud Computing*. IEEE, 522–529. DOI: http://dx.doi.org/10.1109/CLOUD.2014.76

[36] RUBiS 2009. RUBiS: Rice University Bidding System. Retrieved February 20, 2017 from http://rubis.ow2.org/.

[37] Mazeiar Salehie and Ladan Tahvildari. 2009. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems* 4, 2 (May 2009), 1–42. DOI: http://dx.doi.org/10.1145/1516533.1516538

[38] Andrew Symington, Sonia Waharte, Simon Julier, and Niki Trigoni. 2010. Probabilistic target detection by camera-equipped UAVs. In *Proceedings of the 2010 IEEE International Conference on Robotics and Automation*. IEEE, 4076–4081. DOI: http://dx.doi.org/10.1109/ROBOT.2010.5509355

[39] B. Trushkowsky, P. Bodík, A. Fox, and M. J. Franklin. 2011. The SCADS director: Scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*. USENIX Association, 163–176.

[40] Michael J. Veth. 1994. *Advanced Formation Flight Control*. Technical Report. Air Force Institute of Technology.

[41] Eric Yuan, Naeem Esfahani, and Sam Malek. 2014. A systematic survey of self-protecting software systems. *ACM Transactions on Autonomous and Adaptive Systems* 8, 4 (Jan. 2014), 1–41. DOI: http://dx.doi.org/10.1145/2555611

[42] Ji Zhang, Zhenxiao Yang, Betty H. C. Cheng, and Philip K. McKinley. 2004. Adding safeness to dynamic adaptation techniques. In *Proceedings of the ICSE 2004 Workshop on Architecting Dependable Systems*.

[43] Jiheng Zhang and Bert Zwart. 2008. Steady state approximations of limited processor sharing queues in heavy traffic. *Queueing Systems* 60, 3–4 (Nov. 2008), 227–246. DOI: http://dx.doi.org/10.1007/s11134-008-9095-4

Received February 2017; revised August 2017; accepted September 2017