Stochastic Game Analysis and Latency Awareness for Proactive Self-Adaptation

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

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

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

ABSTRACT

Although different approaches to decision-making in self-adaptive systems have shown their effectiveness in the past by factoring in predictions about the system and its environment (e.g., resource availability), no proposal considers the latency associated with the execution of tactics upon the target system. However, different adaptation tactics can take different amounts of time until their effects can be observed. In reactive adaptation, ignoring adaptation tactic latency can lead to suboptimal adaptation decisions (e.g., activating a server that takes more time to boot than the transient spike in traffic that triggered its activation). In proactive adaptation, taking adaptation latency into account is necessary to get the system into the desired state to deal with an upcoming situation. In this paper, we introduce a formal analysis technique based on model checking of stochastic multiplayer games (SMGs) that enables us to quantify the potential benefits of employing different types of algorithms for self-adaptation. In particular, we apply this technique to show the potential benefit of considering adaptation tactic latency in proactive adaptation algorithms. Our results show that factoring in tactic latency in decision making improves the outcome of adaptation. We also present an algorithm to do proactive adaptation that considers tactic latency, and show that it achieves higher utility than an algorithm that under the assumption of no latency is optimal.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General; D.2.4 [**Software/Program Verification**]: Formal methods

General Terms

Verification, Algorithms

Keywords

Proactive adaptation, Stochastic multiplayer games, Latency

SEAMS'14, June 2–3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2864-7/14/06...$15.00
http://dx.doi.org/10.1145/2593929.2593933

1. INTRODUCTION

When planning how to adapt, self-adaptive approaches typically focus on the qualities of the resulting system, such as performance, operating cost, and reliability [15, 29], or safety and liveness properties of the system [4, 16]. However, properties of the adaptation itself are largely ignored. One such property is the time it takes for an adaptation to cause its intended effect. Different adaptation tactics take different amounts of time until their effects can be observed. For example, consider two tactics to deal with an increase in the load of a system: reducing the fidelity of the results (e.g., less resolution, fewer elements, etc.), and adding a computer to share the load. Adapting the system to produce results with less fidelity may be achieved quickly if it can be done by changing a simple setting in a component, whereas powering up an additional computer to share the load may take some time. We refer to the time it takes since a tactic is started until its effect is observed as *tactic latency*. Current approaches to decide how to self-adapt do not take the latency of adaptation tactics into account when deciding what tactic to enact. For reactive adaptation, the consequence of this limitation is that the system may decide to adapt in a way that takes longer than other alternatives to achieve a marginally better result. For proactive adaptation, considering tactic latency is necessary so that the adaptation can be started with the sufficient lead time to be ready in time.

In this paper, we explore the use of tactic latency information in the case of proactive self-adaptation. Specifically, the contribution of this paper is twofold:

1. A novel analysis technique based on model checking of stochastic multiplayer games (SMGs) that enables us to quantify the potential benefits of employing different types of algorithms for self-adaptation. Specifically, we show how the technique enables the comparison of alternatives that consider tactic latency information for proactive adaptation with those that are not latency-aware.

2. A specific latency-aware algorithm for proactive adaptation. The algorithm extends the one that Poladian et al. [26] used to compute the optimal sequence of adaptation decisions for anticipatory dynamic configuration.

Our formal verification results show that factoring in tactic latency in decision making improves the outcome of adaptation both in worst and best-case scenarios. This is consistent with the results obtained for our latency-aware proactive adaptation algorithm, showing that it is able to obtain higher utility than Poladian et al.'s algorithm, which is optimal under the assumption of no tactic latency.

The remainder of this paper is structured as follows: Section 2 summarizes Znn.com, the example used to illustrate our approach. Section 3 introduces some background and related work. Section 4 describes our technique for analyzing adaptation based on model checking of stochastic games. Next, section 5 presents our algorithm for latency-aware proactive adaptation. Finally, section 6 concludes the paper and indicates future research directions.

2. EXAMPLE

Znn.com [9] is a case study portraying a representative scenario for the application of self-adaptation in software systems which has been extensively used to assess different research advances in self-adaptive systems. Znn.com embodies the typical infrastructure for a news website, and has a three-tier architecture consisting of a set of servers that provide contents from backend databases to clients via front-end presentation logic (Figure 1). The system uses a load balancer to balance requests across a pool of replicated servers, the size of which can be adjusted according to service demand. A set of clients makes stateless requests, and the servers deliver the requested contents.

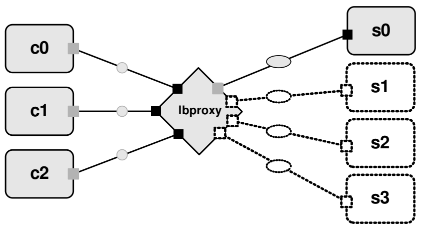

Figure 1: Znn.com system architecture

The main objective for Znn.com is to provide content to customers within a reasonable response time, while keeping the cost of the server pool within a certain operating budget. It is considered that from time to time, due to highly popular events, Znn.com experiences spikes in requests that it cannot serve adequately, even at maximum pool size. To prevent losing customers, the system can maintain functionality at a reduced level of fidelity by setting servers to return only textual content during such peak times, instead of not providing service to some of its customers. Concretely, there are two main quality objectives for the self-adaptation of the system: *(i)* performance, which depends on request response time, server load, and network bandwidth, and *(ii)* cost, which is associated with the number of active servers.

In Znn.com, when response time becomes too high, the system is able to increment its server pool size if it is within budget to improve performance; or switch servers to textual mode if the cost is near to budget limit.[1]

3. BACKGROUND AND RELATED WORK

This section first introduces the adaptation model that we assume in this paper. Next, we overview probabilistic model checking of SMGs, the technique upon which we build to analyze different kinds of adaptation in our approach. Finally, we present related work in proactive self-adaptation.

[1]In this paper we consider a simple version of Znn which adapts only by adjusting server pool size.

3.1 Adaptation Model

Although there are many approaches that rely on a closed-loop control approach to self-adaptation, including those that exploit architectural models for reasoning about the target system under management [15, 20, 25], in this paper we use some of the high-level concepts in Rainbow [15] as a reference framework to illustrate our approach. Rainbow is an architecture-based platform for self-adaptation, which has among its distinct features an explicit architecture model of the target system, a collection of adaptation tactics, and utility preferences to guide adaptation.

We assume a model of adaptation that represents adaptation knowledge using the following high-level concepts:[2]

- **Tactic:** is a primitive action that corresponds to a single step of adaptation, and has an associated: (i) cost/benefit impact on the different quality dimensions, and (ii) latency, which corresponds to the time it takes since a tactic is started until its effect is observed.[3] For instance, in Znn.com we can specify pairs of tactics with opposing effects for enlisting/discharging servers.
- **Utility Profile:** To enable the selection of tactics at run-time, we assume that adaptation is driven by utility functions and preferences, which are sensitive to the context of use and able to consider trade-offs among multiple potentially conflicting objectives. The different qualities of concern are characterized as utility functions that map them to architectural properties. In this case, we assume that utility functions are defined by an explicit set of value pairs (with intermediate points linearly interpolated). Table 1 summarizes the utility functions for Znn. Function $\mathsf{U_R}$ maps low response times (up to 100ms) with maximum utility, whereas values above 2000ms are highly penalized (utility below 0.25), and response times above 4000ms provide no utility. Function $\mathsf{U_C}$ maps a increasing cost (derived from the number of active servers) to lower utility values. Utility preferences capture business preferences over the quality dimensions, assigning a specific weight ($w_{\mathsf{U_R}}$, $w_{\mathsf{U_C}}$) to each one of them. In Znn, we consider that preference is given to performance over cost.

By evaluating how different tactic execution sequences might affect the different qualities of concern using a utility profile, a proactive adaptation algorithm can build a strategy with the objective of maximizing accrued utility throughout the execution of the system.

Table 1: Utility functions and preferences for Znn

$\mathsf{U_R}(w_{\mathsf{U_R}} = 0.6)$			$\mathsf{U_C}(w_{\mathsf{U_C}} = 0.4)$	
0 : 1.00	500 : 0.90	2000 : 0.25	0 : 1.00	3 : 0.30
100 : 1.00	1000 : 0.75	4000 : 0.00	1 : 1.00	4 : 0.00
200 : 0.99	1500 : 0.50		2 : 0.90	

3.2 Model Checking Stochastic Games

Automatic verification techniques for probabilistic systems have been successfully applied in a variety of application domains that range from power management or wireless communication protocols, to biological systems. In particular, techniques such as probabilistic model checking provide a means to model and analyze systems that exhibit stochastic

[2]We use a simplified version of Stitch [11] to illustrate the main ideas in this paper.

[3]Stitch incorporates a different notion of timing delay to monitor the outcome of tactic executions in reactive adaptation strategies, which is not discussed in this paper.

behavior, effectively enabling reasoning quantitatively about probability and reward-based properties (e.g., about the system's use of resources, or time).

Competitive behavior may also appear in (stochastic) systems when some component cannot be controlled, and could behave according to different or even conflicting goals with respect to other components in the system. In such situations, a natural fit is modeling a system as a game between different players, adopting a game-theoretic perspective. Automatic verification techniques have been successfully used in this context, for instance for the analysis of security [21] or communication protocols [19].

Our approach to analyzing adaptation builds upon a recent technique for modeling and analyzing SMGs [6]. In this approach, systems are modeled as turn-based SMGs, meaning that in each state of the model, only one player can choose between several actions, the outcome of which can be probabilistic. Players can either cooperate to achieve the same goal, or compete to achieve their own goals.

The approach includes a logic called rPATL for expressing quantitative properties of stochastic multiplayer games, which extends the probabilistic logic PATL [8]. PATL is itself an extension of ATL [1], a logic extensively used in multiplayer games and multiagent systems to reason about the ability of a set of players to collectively achieve a particular goal. Properties written in rPATL can state that a coalition of players has a strategy which can ensure that either the probability of an event's occurrence or an expected reward measure meets some threshold.

rPATL is a CTL-style branching-time temporal logic that incorporates the coalition operator $\langle\langle C \rangle\rangle$ of ATL [1], combining it with the probabilistic operator $\mathsf{P}_{\bowtie \mathsf{q}}$ and path formulae from PCTL [3]. Moreover, rPATL includes a generalization of the reward operator $\mathsf{R}^{\mathsf{r}}_{\bowtie x}$ from [13] to reason about goals related to rewards. An example of typical usage combining coalition and reward operators is $\langle\langle \{1, 2\} \rangle\rangle \mathsf{R}^{\mathsf{r}}_{\geq 5}[\mathsf{F}^{\mathsf{c}}\phi]$, meaning that "players 1 and 2 have a strategy to ensure that the reward r accumulated along paths leading to states satisfying state formula ϕ is at least 5, regardless of the strategies of other players." Moreover, extended versions of the rPATL reward operator $\langle\langle C \rangle\rangle \mathsf{R}^{\mathsf{r}}_{\mathsf{max}=?}[\mathsf{F}^{\star}\ \phi]$ and $\langle\langle C \rangle\rangle \mathsf{R}^{\mathsf{r}}_{\mathsf{min}=?}[\mathsf{F}^{\star}\ \phi]$, enable the quantification of the maximum and minimum accumulated reward r along paths that lead to states satisfying ϕ that can be guaranteed by players in coalition C, independently of the strategies followed by the rest of players.

Reasoning about strategies is a fundamental aspect of model checking SMGs, which enables checking for the existence of a strategy that is able to optimize an objective expressed as a property including an extended version of the rPATL reward operator. The checking of such properties also supports strategy synthesis, enabling us to obtain the corresponding optimal strategy. An SMG strategy resolves the choices in each state, selecting actions for a player based on the current state and a set of memory elements.[4]

3.3 Related Work

Poladian et al. demonstrated that when there is an adaptation cost or penalty, proactive adaptation outperforms reactive adaptation [26]. Intuitively, if there is no cost associated with adaptation, a reactive approach could adapt at the time a condition requiring adaptation is detected without any negative consequence. In their work, Poladian et al. presented two algorithms for proactive adaptation that considered the penalty of adaptation when deciding how to adapt. One of the algorithms assumed perfect predictions of the environment, while the other handled uncertainty. The latter was used to improve self-adaptation in Rainbow [10], where Cheng et al. considered tactic latency only to skip the adaptation if the condition that triggered it was predicted to go away by itself before the adaptation tactic completed. However, the approach did not consider all the effects that arise due to tactic latency (see Section 5).

Proactive adaptation has received considerable attention in the area of service-based systems [5, 18, 23, 28] because of their reliance on third-party services whose quality of service (QoS) can change over time. In that setting, when a service failure or a QoS degradation is detected, a penalty has already been incurred, for example, due to service-level agreement (SLA) violations. Thus, proactive adaptation is needed to avoid such problems. Hielscher et al. proposed a framework for proactive self-adaptation that uses online testing to detect problems before they happen in real transactions, and to trigger adaptation when tests fail [18]. Wang and Pazat use online prediction of QoS degradations to trigger preventive adaptations before SLAs are violated [28]. These approaches ignore the adaptation latency.

Musliner considers adaptation time by imposing a limit on the time to synthesize a controller for real-time autonomous systems [24]. However, in that work there are not distinct planning and execution phases, and thus there is no consideration of the latency of the different actions the system could take to adapt. In the area of dynamic capacity management for data centers, the work of Gandhi et al. considers the setup time of servers, and is able to deal with unpredictable changes in load by being conservative about removing servers when the load goes down [14]. Their work is specifically tailored to adding and removing servers to a dynamic pool, a setting that resembles the running example we use in this paper. However, their work assumes the environment is unpredictable, and, consequently, does not consider the possibility of being able to predict a reduction or transient spikes in load. Our approach, on the other hand, can exploit predictions of such events and either adapt as soon as possible when removing servers, or avoid adaptations completely.

4. ANALYZING ADAPTATION

This section describes our approach to analyze self-adaptation, based on model checking of SMGs. In a nutshell, the underlying idea behind the approach is modeling both the self-adaptive system and its environment as two players of a SMG, in which the system is trying to maximize an accumulated reward (in the context of this paper, accrued utility while the system is running). Although in general, the environment does not have any predefined goal, it is useful to consider it either as an adversary of the system, or as a cooperative player to enable worst and best-case scenario analysis, respectively, of different classes of adaptation algorithms (e.g., latency-aware *vs.* non-latency-aware).

By expressing properties that enable us to quantify the maximum and minimum rewards that a player can achieve, independently of the strategy followed by the rest of players, we can analyze the performance of a particular type of adaptation algorithm, giving an approximation of the reward that an optimal decision maker would be able to guarantee both

[4]See [6] for more details on SMG strategy synthesis.

in worst and best-case scenarios (by synthesizing strategies that optimize different rewards). These properties follow the general pattern $\langle\langle \mathsf{P} \rangle\rangle \mathsf{R}^{\mathsf{U}}_{\bowtie}[\mathsf{F}^{\mathsf{c}}\omega]$, where P is a set of players that can include the system and/or the environment, U is a reward that encodes the instantaneous utility of the system, $\bowtie \in \{\mathsf{min} =?, \mathsf{max} =?\}$ identifies whether we are considering the minimum or the maximum utility reward, respectively, and ω is a state formula that indicates the termination of the system's execution. Section 4.2 details how such properties are used in our approach.

In the remainder of this section, we first present a SMG model of Znn that enables the comparison of latency-aware against non-latency-aware adaptation. We then describe how these models can be analyzed and show some results for different instances of the model.

4.1 SMG Model

Our formal model is implemented in PRISM-games [7], an extension of the probabilistic model-checker PRISM [22] for modeling and analyzing SMGs. Our game is played in turns by two players that are in control of the behavior of the environment and the system, respectively. The SMG model consists of the following parts:

Player definition. Listing 1 illustrates the definition of the players in the stochastic game: player env is in control of all the (asynchronous) actions that the environment can take (as defined in the environment module), whereas player sys controls all transitions that belong to the target_system module.[5] Global variable turn in line 4 is used to make players alternate, ensuring that for every state of the model, only one player can take action. Turn-based gameplay suffices to naturally model the interplay between the environment and the system, which only senses environment information and reacts to it if necessary at discrete time points.

```
1  player env environment endplayer
2  player sys target_system,[enlist],[enlist_trigger],[discharge] endplayer
3  const ENV_TURN=1, SYS_TURN=2;
4  global turn:[ENV_TURN..SYS_TURN] init ENV_TURN;
```

Listing 1: Player definition for Znn's SMG

Environment. The environment is in control of the evolution of time and other variables of the execution context that are out of the system's control (e.g., service requests arriving at the system). The choices in the environment module are specified non-deterministically to obtain a representative specification of the environment (through strategy synthesis) that is not limited to specific behaviors, since this would limit the generality of our analysis. Listing 2 shows the encoding used for the environment, in which Lines 1-3 define different constants that parameterize its behavior:[6]

- MAX_TIME defines the time frame for the system's execution in the model ([0,MAX_TIME]).
- TAU sets time granularity, defining the frequency with which the environment updates the value of non-controllable variables, and the system responds to these changes. The total number of turns for both players in the SMG is MAX_TIME/TAU. Two consecutive turns of the same player are separated by a time period of duration TAU.
- MAX_ARRIVALS constrains the maximum total number of requests that can arrive at the system for processing throughout its execution. Unconstrained arrivals would result in an unrealistic behavior of the environment (e.g., by following the strategy of continuously flooding the system with requests).
- MAX_INST_ARRIVALS is the maximum number of arrivals that the environment can place for the system to process during its turn (i.e., during one TAU time period).

```
1   const MAX_TIME;
2   const TAU;
3   const MAX_ARRIVALS, MAX_INST_ARRIVALS;
4
5   module environment
6   t : [0..MAX_TIME] init 0;
7   arrivals_total : [0..MAX_ARRIVALS] init 0;
8   arrivals_current : [0..MAX_INST_ARRIVALS] init 0;
9   a_upd : bool init false;
10    [] (t<MAX_TIME) & (turn=ENV_TURN) &
           (arrivals_total+x<MAX_ARRIVALS) & (!a_upd) −>
           (arrivals_current'=x) & (a_upd'=true);
11    ...
12    [] (t<MAX_TIME) & (turn=ENV_TURN) & (a_upd) −>
           1:(t'=t+TAU) & (a_upd'=false) &
           (arrivals_total'=arrivals_total+arrivals_current) &
           (turn'=SYS_TURN);
13  endmodule
```

Listing 2: Environment module

Moreover, lines 6-9 declare the different variables that define the state of the environment:

- t keeps track of execution time.
- arrivals_total keeps track of the accumulated number of arrivals throughout the execution.
- arrivals_current is the number of request arrivals during the current time period.

Each turn of the environment consists of two steps:

1. Setting the amount of request arrivals for the current time period. This is achieved through a set of commands that follow the pattern shown in Listing 2, line 10: the guard in the command checks that (i) it is the turn of the environment to move, (ii) the end of the time frame for execution has not been reached yet, and (iii) the value of request arrivals for the current time period has not been set yet (controlled by flag a_upd). If the guard is satisfied, the command sets the value of request arrivals for the current time period (represented by x in the command). It is worth noticing that there may be as many of these commands as different possible values can be assigned to the number of request arrivals for the current time period (including zero for no arrivals). Probabilities in these commands are left unspecified, since it will be up to the strategy followed by the player (to be synthesized based on an rPATL specification) to provide the discrete probability distribution for this set of commands.
2. Updating the values of the different environment variables (line 12), by: (i) increasing the t time variable one step, and (ii) adding the number of request arrivals for the current time period to the accumulator arrivals_total. In addition, the turn of the environment player finishes when this command is executed, since it modifies the value of variable turn, yielding control to the system player.

[5]Actions enlist_trigger, enlist, and discharge are explicitly labeled to improve readability (see Listing 3), but are still asynchronous in our model.

[6]Constant values not defined in the model are provided as command-line input parameters to the tool.

System. Module target_system (Listing 3) models the behavior of the target system (including the execution of tactics upon it), and is parameterized by the constants:

- MIN_SERVERS and MAX_SERVERS, which specify the minimum and maximum number of active servers that a valid system configuration can have.
- INIT_SERVERS is the number of active servers that the system has in its initial configuration.
- ENLIST_LATENCY is the latency of the tactic for enlisting a server, measured in number of time periods (i.e., the real latency for the tactic in time units is TAU * ENLIST_LATENCY). In our model, tactic latencies are always limited to multiples of the time period duration.
- MAX_RT and INIT_RT, which specify the system's maximum and initial response times, respectively.

```
1  const MIN_SERVERS, MAX_SERVERS, INIT_SERVERS;
2  const ENLIST_LATENCY;
3  const MAX_RT, INIT_RT;
4
5  module target_system
6  s : [0..MAX_SERVERS] init INIT_SERVERS;
7  rt : [0..MAX_RT] init INIT_RT;
8  counter:[−1..ENLIST_LATENCY] init −1;
9     [] (s<=MAX_SERVERS) & (turn=SYS_TURN) & (counter!=0) −>
          (counter'=counter>0?counter−1:counter) &
          (turn'=ENV_TURN) & (rt'=totalTime);
10    [enlist_trigger] (s<MAX_SERVERS) & (turn=SYS_TURN) &
          (counter=−1) −> (counter'=ENLIST_LATENCY) &
          (turn'=ENV_TURN) & (rt'=totalTime);
11    [enlist] (s<MAX_SERVERS) & (turn=SYS_TURN) & (counter=0)
          −> 1: (s'=s+1) & (counter'=−1) & (turn'=ENV_TURN) &
          (rt'=totalTime);
12    [discharge] (s>MIN_SERVERS) & (turn=SYS_TURN) &
          (counter!=0) −> (s'=s−1) &
          (counter'=counter>0?counter−1:counter) &
          (turn'=ENV_TURN) & (rt'=totalTime) ;
13 endmodule
```

Listing 3: System module

Moreover, the module includes variables which are relevant to represent the current state of the system:

- s corresponds to the number of active servers.
- rt is the system's response time.
- counter is used to control the delay between the triggering of a tactic and the moment in which it becomes effective in the target system. In this case, the variable is used to control the delay between the activation of a server, and the time instant in which it really becomes active.

During its turn, the system can decide not to execute any tactics, returning the turn to the environment player by executing the command defined in line 9, Listing 3. Alternatively, the system can execute one of these tactics:

- Activation of a server, which is carried out in two steps:
  1. Triggering of activation through the execution of the command labeled as enlist_trigger (line 10). This command only executes if the current number of active servers has not reached the maximum allowed, and the counter that controls tactic latency is inactive (meaning that there is not currently a server already booting in the system). Upon execution, the command activates the counter by setting it to the value of the latency for the tactic, and returns the turn to the environment player.
  2. Effective activation through the enlist command (line 11), which executes when the counter that controls tactic latency reaches zero, incrementing the number of servers in the system, and deactivating the counter. All the commands in this module, except for the latter, decrement the value of the counter 1 unit, if the counter is activated (counter'=counter>0?counter-1:counter).
- Deactivation of a server, which is achieved through the discharge command (line 12), which decrements the number of active servers. The command fires only if the current number of active servers is greater than the minimum allowed and the counter for server activation is not active.

In addition, all the commands in this module update the value of the response time according to the request arrivals during the current time period and the number of active servers (computed using of an M/M/c queuing model [12], encoded by formula totalTime).

Utility profile Utility functions and preferences are encoded using formulas and reward structures that enable the quantification of instantaneous utility. Specifically, formulas compute utility on the different dimensions of concern, and reward structures weigh them against each other by using the utility preferences.

```
1  formula uR = (rt>=0 & rt<=100? 1:0)
2   +(rt>100&rt<=200?1+(−0.01)*((rt−100)/(100)):0)
3   ...
4   +(rt>2000&rt<=4000?0.25+(−0.25)*((rt−2000)/(2000)):0)
5   +(rt>4000 ? 0:0);
6  ...
7  rewards "rIU"
8   (turn=SYS_TURN) : TAU*(0.6*uR +0.4*uC);
9  endrewards
```

Listing 4: Utility functions and reward structure

Listing 4 illustrates in lines 1-5 the encoding of utility functions using a formula for linear interpolation based on the points defined for utility function U_R in the first column of Table 1. The formula in the example computes the utility for performance, based on the value of the variable for system response time rt. Moreover, lines 7-9 show how a reward structure can be defined to compute a single utility value for any state by using utility preferences. Specifically, each state in which it is the turn of the system player to move is assigned with a reward corresponding to the entire elapsed time period of duration TAU, during which we assume that instantaneous utility does not change.

```
1  rewards "rEIU"
2   (turn=SYS_TURN) : TAU*(0.6*uER +0.4*uC);
3  endrewards
```

Listing 5: Expected utility reward structure

In latency-aware adaptation, the instantaneous real utility extracted from the system coincides with the utility expected by the algorithm's computations during the tactic latency period. However, in non-latency-aware adaptation, the instantaneous utility expected by the algorithm during the latency period for activating a server does not match the real utility extracted for the system, since the new server has not yet impacted the performance. To enable analysis of real *vs.* expected utility in non-latency-aware adaptation, we add to the model a new reward structure that encodes expected instantaneous utility rEIU (Listing 5). In this case, the utility for performance during the latency period (encoded in formula uER) is computed analogously to uR in Listing 4, but based on the response time that the system would have with s+1 servers during the latency period.

4.2 Analysis

In order to compare latency-aware *vs.* non-latency-aware adaptation, we make use of rPATL specifications that enable us to analyze (i) the maximum utility that adaptation can guarantee, independently of the behavior of the environment (worst-case scenario), and (ii) the maximum utility that adaptation is able to obtain under ideal environmental conditions (best-case scenario).

4.2.1 Latency-aware Adaptation

Worst-case scenario analysis. We define the *real guaranteed accrued utility* (U_{rga}) as the maximum real instantaneous utility reward accumulated throughout execution that the system player is able to guarantee, independently of the behavior of the environment player:

$$U_{rga} \triangleq \langle\langle \mathsf{sys} \rangle\rangle \mathsf{R}^{\mathsf{rIU}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{t} = \mathsf{MAX\_TIME}]$$

This enables us to obtain the utility that an optimal self-adaptation algorithm would be able to extract from the system, given the most adverse possible conditions of the environment. Alternatively, U_{rga} can also be obtained by computing a strategy for the environment, based on the minimization of the same reward:

$$\langle\langle \mathsf{env} \rangle\rangle \mathsf{R}^{\mathsf{rIU}}_{\mathsf{min}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{t} = \mathsf{MAX\_TIME}]$$

Best-case scenario analysis. To obtain the *real maximum accrued utility* achievable (U_{rma}), we specify a coalition of the system and environment players, which behave cooperatively to maximize the utility reward:

$$U_{rma} \triangleq \langle\langle \mathsf{sys}, \mathsf{env} \rangle\rangle \mathsf{R}^{\mathsf{rIU}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{t} = \mathsf{MAX\_TIME}]$$

4.2.2 Non-latency-aware Adaptation

In the case of non-latency-aware adaptation, the real utility does not coincide with the expected utility that an arbitrary algorithm would employ for decision-making, therefore we need to proceed with the analysis in two stages:

1. Compute the strategy that the adaptation algorithm would follow based on the information it employs about expected utility. That strategy is computed based on an rPATL specification that obtains the expected guaranteed accrued utility (U_{ega}) for the system player:
   $$U_{ega} \triangleq \langle\langle \mathsf{sys} \rangle\rangle \mathsf{R}^{\mathsf{rEIU}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{t} = \mathsf{MAX\_TIME}]$$
   For the specification of this property we employ the expected utility reward rEIU (Listing 5) instead of the real utility reward rIU. Moreover, it is worth observing that for latency-aware adaptation $U_{ega} = U_{rga}$.
2. Verify the specific property of interest (e.g., U_{rga}, U_{rma}) under the generated strategy. We do this by using PRISM-games to build a product of the existing game model and the strategy synthesized in the previous step, obtaining a new game under which further properties can be verified. In our case, once we have computed a strategy for the system player to maximize expected utility, we quantify the reward for real utility in the new game in which the system player strategy has already been fixed.

4.3 Results

Table 2 compares the results for the utility extracted from the system by a latency-aware *vs.* a non-latency-aware version of the system player, for two different models of Znn that represent an execution of the system during 100 and 200s, respectively. The models consider a pool of up to 4 servers, out of which 2 are initially active. The period duration TAU is set to 10s, and for each version of the model, we compute the results for three variants with different latencies for the activation of servers of up to 3*TAU s. The maximum number of arrivals that the environment can place per time period is 20, whereas the time it takes the system to service every request is 1s.

We define the delta between the expected and the real guaranteed utility as:

$$\Delta U_{er} = (1 - \frac{U_{ega}}{U_{rga}}) \times 100$$

Moreover, we define the delta in real guaranteed utility between latency-aware an non-latency aware adaptation as:

$$\Delta U_{rga} = (1 - \frac{U^n_{rga}}{U^l_{rga}}) \times 100,$$

where U^n_{rga} and U^l_{rga} designate the real guaranteed accrued utility for non-latency-aware and latency-aware adaptation, respectively. The delta in real maximum accrued utility (ΔU_{rma}) is computed analogously to ΔU_{rga}.

Table 2 shows that latency-aware adaptation outperforms in all cases its non-latency-aware counterpart. In the worst-case scenario, latency-aware adaptation is able to guarantee an increment in utility extracted from the system, independently of the behavior of the environment (ΔU_{rga}) that ranges between approximately 10 and 34%, increasing progressively with higher tactic latencies. In the best-case scenario (cooperative environment), the maximum utility that latency-aware adaptation can achieve does not experience noticeable variation with latency, staying in the range 19-23% in all cases. Regarding the delta between expected and real utility that adaptation can guarantee, we can observe that ΔU_{er} is always zero in the case of latency-aware adaptation, since expected and real utilities always have the same value, whereas in the case of non-latency-aware adaptation there is a remarkable decrement that ranges between 24 and 48%, also progressively increasing with higher tactic latency.

5. LATENCY-AWARE ADAPTATION

Latency-aware adaptation takes into account the tactics' latency when deciding how to adapt. In our approach, the goal is to consider the latency of the tactics so that the sum of utility provided by the system over time is maximized. The effect of tactic latency on utility is that for tactics that have some latency, the system does not start to accrue the utility gain associated with the tactic until some time after the enactment of the tactic. Moreover, negative impacts of the tactic may have no latency, and start without delay. For example, when adding a server to the system, the server takes some time to boot and be online, whereas it starts consuming power—and thereby increases cost—immediately. In this example, it means that the tactic to add a server causes a drop in utility before it results in a gain.

Another consequence of tactic latency is that some near-future system configurations can be infeasible. For example, let us suppose that the system has to deal with an increase in load within 5 seconds, and it could handle that with an additional server. If enlisting an additional server takes 10 seconds, then the desired configuration that has one additional server 5 seconds into the future is infeasible. Current approaches that do not take latency into account would consider that solution regardless of whether it is feasible or not. When proactively looking ahead, taking adaptation latency

Table 2: SMG model checking results for Znn

MAX_TIME (s)	Latency (s)	Latency-Aware				Non-Latency-Aware				ΔU_{rga} (%)	ΔU_{rma} (%)
		U_{ega}	U_{rga}	ΔU_{er}(%)	U_{rma}	U_{ega}	U_{rga}	ΔU_{er}(%)	U_{rma}		
100	TAU	53.77	53.77	0	99.6	65.97	48.12	-27.05	79.99	10.5	19.68
	2*TAU	49.35	49.35	0	99.6	64.3	42.1	-34.5	78.39	14.69	21.29
	3* TAU	45.6	45.6	0	99.6	64.3	33.25	-48.2	78.39	27	21.29
200	TAU	110.02	110.02	0	199.6	127.25	95.9	-24.63	156.79	12.83	21.44
	2*TAU	105.6	105.6	0	199.6	125.57	76.6	-38.99	155.19	27.46	22.24
	3* TAU	101.17	101.17	0	199.6	123.9	66.15	-46.6	153.59	34.61	23.05

into account allows the adaptation mechanism to rule out infeasible configurations from the adaptation space.

A complication arises when tactic latency is longer than the interval between adaptation decisions. When that is the case, it is possible that during an adaptation decision, a tactic that has been previously started has not yet reached the point where its effect will have been realized. If the decisions are made based only on the currently observed state of the system, ignoring the expected effect of adaptations in progress, the system will overcompensate, starting unnecessary adaptations. What is needed is a model of the system that not only represents the current state of the system, but also keeps track of the expected state of the system in the near future based on the tactics that have been started but have not yet completed.

5.1 Algorithm

The algorithm we present is an extension of an algorithm developed by Poladian et al. to compute the optimal sequence of adaptation decisions for anticipatory dynamic configuration [26]. Using dynamic programming and relying on a perfect prediction of the environment for the duration of a system run, their algorithm can find the adaptation decision that at each time step maximizes the future utility, while accounting for the penalty of switching configurations. They showed that the algorithm had pseudo-polynomial time complexity, and was therefore suitable for online adaptation.

The key improvement our algorithm brings is how the latency of tactics is taken into account. On the one hand, there is an adaptation cost that latency induces. For example, if adding a server takes λ seconds from the time a server is powered up until it can start processing requests, and ΔU_c is the additional cost the new server incurs, then the adaptation cost is $\lambda \Delta U_c$. This cost could be partially handled by the original algorithm, as a reconfiguration penalty. However, that is not sufficient to handle the other issues previously mentioned that latency brings, namely, the infeasibility of configurations and the need to track adaptation progress. Our algorithm for latency-aware proactive adaptation (Algorithm 1) explicitly handles the issues that arise due to tactic latency.

In reactive adaptation, the decision algorithm is typically invoked upon events that require an adaptation to be performed. However, for proactive adaptation, the decision must be done periodically, looking ahead for future states that may require the system to adapt. This algorithm is therefore run periodically, with a constant interval between runs. We limit the look-ahead of the algorithm to a near-term horizon, which in turn limits how far into the future the environment state needs to be estimated.[7]

The algorithm relies on these functions and variables:

[7]Environment state estimation is beyond the scope of our work, but techniques such as Poladian et al.'s calculus for combining multiple source of predictions [26] can be used.

- C is the set of possible configurations, and C_i is the ith configuration, for $i \in [1 \ldots |C|]$.
- $servers(c)$ is the number of active servers (i.e., servers that can process requests) in configuration c.
- $totalServers(c)$ is the total number of servers in configuration c, including active servers and servers that have been powered up but are not active yet.
- λ is the amount of time it takes for a server to become active after being powered up.
- $sys(x)$ is the expected system configuration x time units into the future. This function is used to query the model of the system that keeps track of adaptations in progress to project what is the expected system configuration in the near future. The current system configuration can be obtained with $sys(0)$.
- $env(x)$ is the expected environment state x time units into the future.
- τ is the length of evaluation period.
- H is the look-ahead horizon in terms of evaluation periods. It is required that $\tau H \geq \lambda$ so that the algorithm is able to evaluate the utility after a new server becomes active.
- $U(c, e)$ is the instantaneous utility provided by configuration c in environment e.
- $\Delta U_c(i, j)$ is the difference in cost (negative utility) experienced when changing from a configuration with i servers to one with j servers.

To do dynamic programming, the algorithm uses two matrices, u and n, to store partial solutions. The element $u_{i,t}$ holds the utility projected to be achieved from the evaluation period t (with $t = 0$ being the current period, $t = 1$ the next one, and so on) until the horizon if the system has configuration C_i at evaluation period t (a value of $-\infty$ is used to represent infeasible solutions). The element $n_{i,t}$ holds the configuration that the system must adopt in period $t + 1$ to attain the projected utility $u_{i,t}$ if the configuration at time t is C_i. The loop in lines 1-4 initialize the elements of these matrices at the horizon. In this case, the projected utility of a configuration is the utility that configuration would achieve given the state of the environment predicted at the horizon. The following loop (lines 5-29) works backwards from the horizon, computing the partial solutions using the partial solutions previously found. For each configuration (lines 6-28), it computes its projected utility or deems the configuration infeasible. At any given time, a configuration is feasible if either the system is expected to have enough active servers at that time, or if there is enough time to add the needed servers (line 9). For a feasible solution, the projected utility it can achieve is the sum of the utility the configuration obtains in that particular evaluation period (line 10), and the maximum utility it can achieve in the periods after that, taking into account any adaptation costs. To compute the latter, the algorithm iterates over all the feasible configurations that can follow (lines 12-26) to find the adaptation that maximizes the projected utility (lines 21-24). The adaptation cost incurred for going from configuration C_i in period t to C_j in period $t+1$ is computed in lines 14-19. To do so, we must determine how many active servers will already be available in period t, and find the cost increase, if any, to get to the number of active servers needed by configuration C_j. In general, the number of active servers available in period t, which is the number of servers required by configuration C_i, will be carried over to period $t+1$ if needed because they will already be active. However, if more servers are expected to be active in period $t+1$, i.e., in the expected system configuration ($sys((t+1)\tau)$), we can assume that there will be that number of active servers (line 15), as long as not enough time will have passed to allow the decision of removing a server (line 14).[8]

Once all the possible solutions have been computed, the algorithm searches for the configuration the system should have at the current time to maximize the projected utility (line 30). Finally, it determines if more servers need to be added now so that they are active by the time they are needed. It does so by looking at the sequence of configurations that should be adopted in the following evaluation periods (lines 32-40). The algorithm returns the number of servers that should be added to (if positive) or removed from (if negative) the system (line 41), taking into account the latency of the adaptation tactics.

5.2 Simulation

We implemented a simulation of a self-adaptive Znn with two goals. One was to evaluate the improvement that our algorithm for latency-aware (LA) proactive adaptation achieves compared to a non-latency-aware (NLA) approach. The second one, was to compare the theoretical results obtained with the SMG for generic NLA and LA algorithms with the results obtained with a concrete algorithm. Using simulation allowed us to run many repetitions of the experiments with randomly generated behaviors of the environment.

The simulation was implemented using OMNeT++, an extensible discrete event simulation environment [27]. It simulates the arrival of requests from clients, randomly generating requests. The requests arrive at the load balancer of Znn, and are forwarded to one of the idle servers. If no server is idle, then the requests are queued in FIFO order until one server becomes available. Each server processes one request at a time, with a service time distributed with an exponential distribution with a rate of 1.

The inter-arrival times between client requests are generated randomly with a rate that changes periodically, matching the possibilities of the environment in the SMG. Every τ units of time, a new arrival rate is selected randomly from the interval $[0, 2]$ with a uniform distribution. That rate is then used to generate exponentially distributed inter-arrivals until the next rate is selected. To be able to simulate the execution of the system with the same random pattern of client requests using each of the two algorithms, the request inter-arrival times and the service times are drawn from two

[8]When planning ahead, we assume that a server will not be removed before it becomes active (that is, λ units of time after it was added), otherwise, adding it in the first place would have made no sense. However, we do consider the worst case of a server being removed after having been active for just one evaluation period.

Algorithm 1 Latency-aware proactive adaptation

```
 1: for all $i \in [1 \ldots |C|]$ do
 2:     $u_{i,H} \leftarrow \tau U(C_i, env(\tau H))$
 3:     $n_{i,H} \leftarrow 0$   // no next state
 4: end for
 5: for $t = H - 1$ downto $0$ do
 6:     for all $i \in [1 \ldots |C|]$ do
 7:         $u_{i,t} \leftarrow -\infty$   // assume infeasible configuration
 8:         $n_{i,t} \leftarrow 0$
 9:         if $servers(C_i) \leq servers(sys(t\tau)) \vee \lambda \leq t\tau$ then
10:             $u_{local} \leftarrow \tau U(C_i, env(t\tau))$
11:             // find the next best configuration after i
12:             for all $j \in [1 \ldots |C|]$ do
13:                 if $u_{j,t+1} > -\infty$ then
14:                     if $t\tau < \lambda$ then
15:                         $start \leftarrow \max(servers(C_i), servers(sys((t+1)\tau)))$
16:                     else
17:                         $start \leftarrow servers(C_i)$
18:                     end if
19:                     $cost \leftarrow \max(0, \lambda \Delta U_c(start, servers(C_j))$
20:                     $u_{projected} \leftarrow u_{j,t+1} + u_{local} - cost$
21:                     if $u_{projected} > u_{i,t}$ then
22:                         $u_{i,t} \leftarrow u_{projected}$
23:                         $n_{i,t} \leftarrow j$
24:                     end if
25:                 end if
26:             end for
27:         end if
28:     end for
29: end for
30: $best \leftarrow \arg\max_i u_{i,0}$   // best starting configuration
31:     // find if there is a config with more servers that must be started now
32: $i \leftarrow best$
33: $t \leftarrow 0$
34: while $t < H \wedge (t+1)\tau \leq \lambda$ do
35:     $i \leftarrow n_{i,t}$
36:     if $servers(C_i) > servers(C_{best})$ then
37:         $best \leftarrow i$
38:     end if
39:     $t \leftarrow t + 1$
40: end while
41: return $servers(C_{best}) - totalServers(sys(0))$
```

separate random number generators. Thus, we can compare the utility each algorithm achieves when the system faces the same pattern of client requests.

The self-adaptive layer of the simulated system works as follows. The system is monitored by keeping track of request inter-arrival times when a client request arrives, and of the request response times every time a request processing completes. Once every evaluation interval τ, these observations are used to compute their average and standard deviation for the period since the last evaluation. Using the average response time, and the number of servers in the system, the utility accrued since the last evaluation is computed using the utility functions and preferences shown in Table 1.

Next, the adaptation algorithm is used to determine if the system should self-adapt and how. We implemented both the latency-aware algorithm (Algorithm 1) and a non-latency-aware algorithm. The latter is basically the same as the former, except that it does not account for latency other than by considering the adaptation penalty induced by the cost of having a server powered until becomes active. Indeed, the NLA algorithm can be obtained by replacing all the occurrences of λ in Algorithm 1, except for the one in line 19, with 0. Since the SMG model can only handle the addition or removal of one server at a time, the implementation of the algorithms were modified to adhere to that limitation so that the results were comparable.

The $sys(x)$ function used by the algorithms was implemented by maintaining a model of the system configuration that keeps track of the number of servers in the system, and how many of them are active. In addition, the model keeps a list of expected changes in the future. For example, when a new server is added to the system, an expected change reflecting that the server becomes active is recorded with an expected time of λ into the future. When $sys(x)$ is invoked, the expected system state at x time units into the future can be obtained by taking the current system configuration and applying all the changes expected for the following x time units. When a server actually becomes active in the simulation, the model of the current system configuration is updated to reflect that change and the corresponding entry is removed from the list of expected system changes.

The predictive model of the environment, $env(x)$ was implemented as an oracle that can predict perfectly the average and variance of the request inter-arrival times for the same horizon used by the algorithm. Although the request arrivals are randomly generated in the simulation, a perfect prediction can still be achieved by generating the inter-arrival times before they are consumed by the simulation.

Implementing the $U(c, e)$ function requires first estimating the average response time for requests when the system has configuration c, and the environment is e. In this case, the relevant properties of the environment are the average and variance of the inter-arrival times. To estimate the average response time needed for the utility calculation, we used queueing theory with a $G/M/c$ queueing model (i.e., for arrivals with a general distribution,[9] exponentially distributed service times, and s servers) [17]. Once the average response time is estimated in this way, the utility is estimated using the utility functions and preferences shown in Table 1.

After the adaptation algorithm has determined how the system has to be changed, the execution of the adaptation tactics is carried out by adding or removing servers as needed. The standard queuing components of OMNET++ were modified to support this dynamic reconfiguration. Furthermore, the server component was modified to simulate the latency of enlisting a server.

5.3 Results

We ran the simulation with the same parameters used for the SMG analysis, as described in 4.3. The horizon used for the algorithms was computed so that if the system was running with one server, it had a horizon large enough to be able to compute the effect of adding the three remaining servers. For that reason, the horizon was calculated as $3\frac{\lambda}{\tau} + 1$, the number of periods needed to enlist three servers plus one more period to consider the impact on utility of the change. For each combination of parameters, the simulation was run 1000 times to obtain the statistics shown in Table 3. On average, the latency-aware algorithm outperformed the non-latency-aware one. The LA algorithm obtained on average about 5% more utility when the tactic latency was equal to the evaluation period, and 10% for latencies two and three times larger than the evaluation period. The standardized effect size measure statistic $\hat{A}_{12}$ [2] shows that LA outperforms NLA 66% to 81% of the times, depending on the parameters. For several combinations of parameters, the minimum percentual utility difference $\Delta U(\%)$ was negative, meaning that NLA did better. This is due to a limitation of the queueing model used by the algorithms to estimate the response time of different configurations, because it computes the steady-state response time, and, therefore, ignores the effect of arrival spikes that may leave a backlog of arrivals to be processed in later periods. The LA algorithm avoids adaptation when there are transient increases in load if the cost of enlisting a server will be higher than the negative impact of not adding it. Because of the limitation of the queueing model, it sometimes underestimates that negative effect. Since the NLA algorithm does not account for the latency of the tactic, it is more prone to add servers, and that gives it an advantage in these cases. These situations were not very common in our experiment runs, as indicated by the 10% quantile, which, except for the cases with the lowest tactic latency, was positive. Furthermore, it is worth noting that this is a limitation of the $U(c, e)$ function used by the algorithm, and not a problem with the algorithm itself.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have described an analysis technique based on model checking of stochastic multiplayer games that enables the quantification of the potential benefits that different types of algorithms for decision making in adaptation can yield. We have shown how this technique can be used in the context of comparing proactive adaptation algorithms that consider information about tactic latency for decision-making, against those that do not account for it. We have used Znn.com to illustrate our approach.

Our results show that latency-aware proactive adaptation always performs better than non-latency-aware adaptation both in the worst and best-case scenarios, with progressively increasing improvements with higher tactic latencies.

A current limitation of the approach is that its scalability is limited by PRISM-games, which currently uses explicit-state data structures and is to the best of our knowledge the only tool supporting model-checking of SMGs. In our case, the largest SMG model employed for Znn has of the order of 10^6 states, whereas the results presented in [7] show that the current version of the tool can handle models of up to 10^7 states in a common desktop PC. However, the authors of PRISM-games are developing a symbolic (BDD-based) version of the tool that will improve scalability.

We have also proposed a latency-aware proactive adaptation algorithm that is able to exploit predictions about the future behavior of the environment. We have compared our algorithm against the proactive algorithm presented in [26], which does not consider latency, showing that latency-aware adaptation achieves higher utility.

Regarding future work, we plan to instantiate our adaptation analysis technique in different contexts. In particular, we are working on applying this approach to self-protecting systems, studying how different adaptation alternatives can minimize the damage that an attacker can inflict. We also aim at refining the approach to do run-time synthesis of proactive adaptation strategies based on SMGs. Concerning latency-aware adaptation, we aim at exploring how tactic latency information can be further exploited to attain better results both in proactive and reactive adaptation (e.g., by

[9]We chose to use a model for a general distribution of arrivals since: (i) although arrivals are generated with an exponential distribution, the rate parameter of the distribution is changed periodically, and (ii) the queueing model is for steady-state behavior and does not account for any backlog of requests that could have remained in the system from a previous period with higher traffic intensity. Hence, we found the general distribution model was a better fit.

Table 3: Simulation results for Znn

MAX_TIME	Latency	Latency-Aware			Non-Latency-Aware				$\Delta U(\%)$			
(s)	(s)	min.	avg.	max.	min.	avg.	max.	$\hat{A}_{12}$	min.	10% quant.	avg.	max.
100	TAU	39.18	67.29	84.41	33.80	62.63	84.49	0.66	-27.15	-0.65	6.73	31.32
	2*TAU	44.66	69.33	84.55	36.33	62.31	83.20	0.73	-23.86	3.10	10.34	37.69
	3* TAU	48.05	69.40	84.55	31.14	62.48	83.20	0.72	-0.88	3.12	10.24	38.66
200	TAU	81.99	133.20	167.20	82.48	125.00	156.90	0.69	-15.63	-0.96	5.98	21.70
	2*TAU	105.90	138.10	167.20	80.46	124.40	160.00	0.81	-7.82	4.89	10.05	30.53
	3* TAU	106.20	138.40	167.20	85.81	124.70	160.00	0.81	0.00	4.85	10.01	28.32

parallelizing tactic executions). We will also generalize the algorithm to consider multiple tactics with different latency, as well as prediction and tactic latency uncertainty. Moreover, we will implement our algorithms in Rainbow/Znn.

7. ACKNOWLEDGMENTS

This work is supported in part by awards W911NF-13-1-0154 from the Army Research Office and N000141310401 from the Office of Naval Research. Co-financed by the Foundation for Science and Technology via project CMU-PT/ELE/0030/2009 and by FEDER via the "Programa Operacional Factores de Competitividade" of QREN with COMPETE ref.: FCOMP-01-0124-FEDER-012983. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. (DM-0001017).

8. REFERENCES

[1] R. Alur et al. Alternating-time temporal logic. *J. ACM*, 49(5), 2002.

[2] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 2012.

[3] A. Bianco and L. de Alfaro. Model checking of probabalistic and nondeterministic systems. In *FSTTCS*, volume 1026 of *LNCS*. Springer, 1995.

[4] V. Braberman et al. Controller synthesis: From modelling to enactment. In *ICSE*. IEEE, 2013.

[5] R. Calinescu et al. Dynamic QoS Management and Optimization in Service-Based Systems. *IEEE Trans. Software Eng.*, 37(3), 2011.

[6] T. Chen et al. Automatic verification of competitive stochastic systems. *Form Method Syst Des*, 43(1), 2013.

[7] T. Chen et al. PRISM-games: A model checker for stochastic multi-player games. In *Proc. of TACAS'13*, volume 7795 of *LNCS*. Springer, 2013.

[8] T. Chen and J. Lu. Probabilistic alternating-time temporal logic and model checking algorithm. In *FSKD*, volume 2, 2007.

[9] S. Cheng et al. Evaluating the Effectiveness of the Rainbow Self-Adaptive System. In *SEAMS*. IEEE, 2009.

[10] S.-W. Cheng et al. Improving architecture-based self-adaptation through resource prediction. In *SEfSAS*, volume 5525 of *LNCS*. Springer, 2009.

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

[12] R. Chiulli. *Quantitative Analysis: An Introduction.* Automation and production systems. Taylor & Francis, 1999.

[13] V. Forejt et al. Automated verification techniques for probabilistic systems. In *SFM*, volume 6659 of *LNCS*. Springer, 2011.

[14] A. Gandhi et al. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.*, 30(4), 2012.

[15] D. Garlan et al. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer*, 37(10), 2004.

[16] R. Goldman et al. Managing online self-adaptation in real-time environments. In *Self-Adaptive Software: Applications*, volume 2614 of *LNCS*. Springer, 2003.

[17] D. Gross et al. *Fundamentals of Queueing Theory.* Wiley Series in Probability and Statistics. Wiley, 2011.

[18] J. Hielscher et al. A framework for proactive self-adaptation of service-based applications based on online testing. volume 5377 of *LNCS*. Springer, 2008.

[19] W. V. D. Hoek and M. Wooldridge. Model checking cooperation, knowledge, and time - a case study. In *Research in Economics*, 2003.

[20] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE*, 2007.

[21] S. Kremer and J.-F. Raskin. A game-based verification of non-repudiation and fair exchange protocols. In *CONCUR 2001*, volume 2154 of *LNCS*. Springer, 2001.

[22] M. Kwiatkowska et al. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *LNCS*. Springer, 2011.

[23] A. Metzger et al. Accurate proactive adaptation of service-oriented systems. In *ASAS*, volume 7740 of *LNCS*. Springer, 2013.

[24] D. Musliner. Imposing real-time constraints on self-adaptive controller synthesis. In *Self-Adaptive Software*, volume 1936 of *LNCS*. Springer, 2001.

[25] P. Oreizy et al. An architecture-based approach to self-adaptive software. *IEEE Intell. Syst.*, 14, 1999.

[26] V. Poladian et al. Leveraging resource prediction for anticipatory dynamic configuration. In *SASO*, 2007.

[27] A. Varga et al. An overview of the OMNeT++ simulation environment. In *Simutools*. ICST, 2008.

[28] C. Wang and J.-L. Pazat. A two-phase online prediction approach for accurate and timely adaptation decision. In *SCC*, 2012.

[29] X. Zhang and C.-H. Lung. Improving software performance and reliability with an architecture-based self-adaptive framework. In *COMPSAC*, 2010.