Optimal Planning for Architecture-Based Self-Adaptation Via Model Checking of Stochastic Games

Javier Cámara, David Garlan, Bradley Schmerl, Ashutosh Pandey
Institute for Software Research, Carnegie Mellon University
Pittsburgh, PA 15213, USA
{jcmoreno, garlan, schmerl, ashutoshp}@cs.cmu.edu

ABSTRACT

Architecture-based approaches to self-adaptation rely on architectural descriptions to reason about the best way of adapting the structure and behavior of software-intensive systems at runtime, either by choosing among a set of predefined adaptation strategies, or by automatically generating adaptation plans. Predefined strategy selection has a low computational overhead and facilitates dealing with uncertainty (e.g., by accounting explicitly for contingencies derived from unexpected outcomes of actions), but requires additional designer effort regarding the specification of strategies and is unable to guarantee optimal solutions. In contrast, runtime plan generation is able to explore a richer solution space and provide optimal solutions in some cases, but is more limited when dealing with uncertainty, and incurs higher computational overheads. In this paper, we propose an approach to optimal adaptation plan generation for architecture-based self-adaptation via model checking of stochastic multiplayer games (SMGs). Our approach enables: (i) trade-off analysis among different qualities by means of utility functions and preferences, and (ii) explicit modeling of uncertainty in the outcome of adaptation actions and the behavior of the environment. Basing on the concepts embodied in the Rainbow framework for self-adaptation, we illustrate our approach in Znn.com, a case study that reproduces the infrastructure for a news website.

Categories and Subject Descriptors

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

Keywords

Self-adaptation, Planning, Probabilistic Model Checking

1. INTRODUCTION

During the last decade, advances in self-adaptive systems [12, 14, 15, 17] have addressed the rising development and operation costs, as well as the reduction in reliability of increasingly complex software-intensive systems that operate in dynamically changing environments. Specifically, architecture-based self-adaptation approaches [12, 15, 17] rely on architectural descriptions to reason at runtime about the best way to adapt the structure and behavior of a software system to changes in its environment (e.g., resource availability), requirements, and the system itself (e.g., faults).

Architecture-based self-adaptation approaches address adaptation planning mainly in two different ways. On the one hand, approaches that rely on selection of adaptation strategies defined by a designer at development time [12, 17] have a low runtime overhead, often enable candidate solution ranking by analyzing trade-offs among different quality concerns, and are sometimes able to deal with some aspects of uncertainty [12]. However, these approaches are limited to a restricted solution space (hence being unable to guarantee optimal solutions) and require additional designer effort to specify adaptation strategies. On the other hand, approaches that automatically generate adaptation plans at runtime [18, 19] free the designer from specifying strategies at development time, but incur a significantly higher computational overhead compared to predefined strategy selection approaches. Moreover, although some of the latter approaches are able to rank candidate solutions by analyzing trade-offs among different qualities [18] or consider uncertainty for tuning the operation of the system (e.g., by dynamically adjusting parameters [4, 10]), there is no approach to the best of our knowledge able to factor in all these elements for adaptation plan generation.

In this paper, we contribute a novel approach to automatically synthesize optimal reactive adaptation plans via model checking of stochastic multiplayer games (SMGs) [6] that enables: (i) trade-off analysis among different qualities by means of utility functions and preferences, and (ii) modeling of uncertainty both as probabilistic outcomes of adaptation actions and through explicit modeling of the behavior of the system's environment.

We employ some of the concepts in Rainbow [12] as a reference framework to illustrate our approach in the context of Znn.com, a benchmark case study that reproduces the typical infrastructure for a news website.

Our results show a reasonable scalability for plan synthesis when it just considers probabilistic outcomes of actions. Although scalability is more limited for large problem instances that consider the environment's behavior, the richer modeling of uncertainty reduces the need to replan when action executions deviate from their expected outcome.

SAC'15 April 13-17, 2015, Salamanca, Spain.

http://dx.doi.org/xx.xxxx/xxxxxxx.xxxxxxx

In the remainder of the paper, Section 2 presents Znn.com. Section 3 introduces some background on SMGs and the adaptation model that we assume. Section 4 describes the approach. Section 5 presents results, and Section 6 describes related work. Section 7 discusses conclusions and future work.

2. EXAMPLE

Znn.com [8] is a case study portraying a representative scenario for the application of self-adaptation in software systems. It has been extensively used to assess different research advances in self-adaptive systems. Znn.com embodies the typical infrastructure for a news website, and its architecture includes 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.

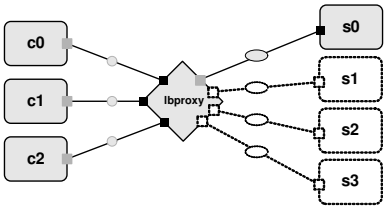

Figure 1: Znn.com system architecture

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. Znn.com can experience 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 three main quality objectives for the self-adaptation of the system: *(i)* performance, which depends on request response time, server load, and network bandwidth, *(ii)* cost, which is associated with the number of active servers, and *(iii)* the fidelity of the contents served.

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.

We assume that there is information available regarding the reliability of the different servers in the pool (derived from observations of previous system executions). This information can be exploited to prioritize the activation of the most reliable servers when trying to increase performance.

3. BACKGROUND

This section introduces the adaptation model that we assume in this paper, and overviews probabilistic model checking of SMGs, the technique on which we build our approach.

3.1 Adaptation Model

Although there are many proposals that rely on a closed-loop control approach to self-adaptation that exploit architectural models for adaptation [12, 15, 17], in this paper we use some of the high-level concepts in Rainbow [12] as a reference framework to illustrate our approach. Rainbow 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:[1]

- **Tactic:** is a primitive action that corresponds to a single step of adaptation, and has an associated cost/benefit impact on the different quality dimensions. For instance, in Znn.com we can specify pairs of tactics with opposing effects for enlisting/discharging servers and raising/lowering the fidelity level of contents served. Table 1 shows the different tactics in Znn, and their impacts on quality dimensions: enlistServer adds a new server, reducing response time and increasing cost, whereas dischargeServer has the opposite effect. In contrast, lowerFidelity/raiseFidelity marginally impact cost, having a moderate impact on response time (compared to server enlisting/discharging) at the expense of lowering fidelity.

Table 1: Cost/benefit for Znn.com tactics.

Tactic	ΔResponse time(ms)	ΔCost(usd/hr)	ΔFidelity level
enlistServer	-1000	+1.0	0
dischargeServer	+1000	-1.0	0
lowerFidelity	-500	-0.1	-1
raiseFidelity	+500	+0.1	+1

- **Utility Profile:** To enable the selection of tactics at runtime, 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 2 summarizes an example of 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. Function $\mathsf{U_C}$ maps higher cost (derived from the number of active servers) to lower utility values. Function $\mathsf{U_F}$ maps a low fidelity level to a utility value of 0.5, whereas maximum fidelity level yields maximum utility. Utility preferences capture business preferences over the quality dimensions, assigning a weight to each one of them (e.g., $w_{\mathsf{U_R}} = 0.5$, $w_{\mathsf{U_C}} = 0.3$, and $w_{\mathsf{U_F}} = 0.2$ indicate performance as the main concern, followed by cost).

Table 2: Utility functions and preferences.

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

3.2 Model Checking Stochastic Games

Probabilistic model checking provides a means to model and analyze systems that exhibit stochastic behavior, enabling quantitative reasoning about probability and reward-based properties (e.g., resource usage, time, etc.).

Competitive behavior may also appear in (stochastic) systems when some component cannot be controlled, and could

[1]We use a simplified version of Stitch [9] to illustrate the main ideas in this paper.

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. This perspective is particularly useful in self-adaptive systems, since the environment can be modeled as a player whose actions cannot be controlled by the system.

Our approach to synthesizing adaptation plans builds upon a recent technique for modeling and analyzing stochastic multi-player games (SMGs) [6] where systems are modeled as turn-based games. Hence, in each state of the model, only one player can choose between several actions, the outcome of which can be probabilistic.

DEFINITION 1 (SMG). *A turn-based SMG is a tuple* $\mathcal{G} = \langle \Pi, S, A, (S_i)_{i\in\Pi}, \Delta, AP, \chi, r\rangle$, *where* Π *is a finite set of players;* $S \neq \emptyset$ *is a finite set of states;* $A \neq \emptyset$ *is a finite set of actions;* $(S_i)_{i\in\Pi}$ *is a partition of* S*;* $\Delta : S \times A \to \mathcal{D}(S)$ *is a (partial) transition function;* AP *is a finite set of atomic propositions;* $\chi : S \to 2^{AP}$ *is a labeling function; and* $r : S \to \mathbb{Q}_{\geq 0}$ *is a reward structure mapping each state to a non-negative rational reward.* $\mathcal{D}(X)$ *denotes the set of discrete probability distributions over finite set* X*.*

In each state $s \in S$, the set of available actions is denoted by $A(s) = \{a \in A \mid \Delta(s, a) \neq \bot\}$. We assume that $A(s) \neq \emptyset$ for all states. Moreover, the choice of which action to take in every state s is under the control of a single player $i \in \Pi$, for which $s \in S_i$. Once a player selects action a, the successor state is chosen according to probability distribution $\Delta(s, a)$.

DEFINITION 2 (PATH). *A path of SMG* $\mathcal{G}$ *is an (in)finite sequence* $\lambda = s_0 a_0 s_1 a_1 \ldots$ s.t. $\forall j \in \mathbb{N} \bullet a_j \in A(s_j) \wedge \Delta(s_j, a_j)(s_{j+1}) > 0$. $\Omega^+_{\mathcal{G}}$ *denotes the set of finite paths in* $\mathcal{G}$*.*

Players in the game can follow strategies for choosing actions in the game, cooperating with each other in coalition to achieve a common goal, or competing to achieve their own (potentially conflicting) goals.

DEFINITION 3 (STRATEGY). *A strategy for player* $i \in \Pi$ *in* $\mathcal{G}$ *is a function* $\sigma_i : (SA)^* S_i \to \mathcal{D}(A)$ *which, for each path* $\lambda \cdot s \in \Omega^+_{\mathcal{G}}$ *where* $s \in S_i$*, selects a probability distribution* $\sigma_i(\lambda \cdot s)$ *over* $A(s)$*.*

In this paper, we always refer to strategies σ_i that are *memoryless* (i.e., $\sigma_i(\lambda \cdot s) = \sigma_i(\lambda' \cdot s)$ for all paths $\lambda \cdot s$, $\lambda' \cdot s \in \Omega^+_{\mathcal{G}}$), and *deterministic* (i.e., $\sigma_i(\lambda \cdot s)$ is a Dirac distribution for all $\lambda \cdot s \in \Omega^+_{\mathcal{G}}$). Memoryless, deterministic strategies resolve the choices in each state $s \in S_i$ for player $i \in \Pi$, selecting actions based solely on information about the current state in the game. These strategies are guaranteed to achieve optimal expected rewards for the kind of cumulative reward structures that we use in our models.[2]

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 quantitative property in a logic called rPATL, which extends ATL [1], a logic extensively used 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 the probability of an event's occurrence or an expected reward measure meet some threshold.

rPATL is a CTL-style branching-time temporal logic that incorporates the coalition operator $\langle\langle C\rangle\rangle$ of ATL, combining it with the probabilistic operator $\mathsf{P}_{\bowtie \mathsf{q}}$ and path formulae from PCTL [2]. Moreover, rPATL includes a generalization of the reward operator $\mathsf{R}^{\mathsf{r}}_{\bowtie \mathsf{x}}$ [11] 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 accrued 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 $\langle\langle C\rangle\rangle \mathsf{R}^{\mathsf{r}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \phi]$ and $\langle\langle C\rangle\rangle \mathsf{R}^{\mathsf{r}}_{\mathsf{min}=?}[\mathsf{F}^{\mathsf{c}}\ \phi]$, enable the quantification of the maximum and minimum accrued 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.

Model checking of rPATL properties supports optimal strategy synthesis for a given property. In the following section, we show how model checking of rPATL reward-based properties can be used to generate optimal adaptation plans.

4. APPROACH

This section describes our approach to planning via model checking of SMGs. In a nutshell, the approach is based on modeling the self-adaptive system and its environment as two players of a SMG, in which the system's objective is reaching a goal state that maximizes a (utility) reward.

By expressing properties that enable us to quantify the maximum reward that a player can achieve, independently of the strategy followed by the rest of players, we can synthesize the corresponding optimal strategy that a decision maker would follow to maximize that reward.

The approach consists of two stages:

1. **SMG Model Specification** of the solution space as a SMG $\mathcal{G} = \langle \Pi, S, A, (S_i)_{i\in\Pi}, \Delta, AP, \chi, r\rangle$, where:
   - $\Pi = \{sys, env\}$ is the set of players formed by the self-adaptive system and its environment.
   - $S = S_{sys} \cup S_{env}$ is the set of states, where S_{sys} and S_{env} are the states controlled by the system and the environment players, respectively ($S_{sys} \cap S_{env} = \emptyset$).
   - $A = A_{sys} \cup A_{env}$ is the set of actions, where A_{sys} and A_{env} are the actions available to the system and the environment players, respectively.
   - AP is a subset of all the predicates that can be built over the state variables. AP always includes:
     - *goal*, which is satisfied in states where the goal of adaptation is achieved.
     - *end*, which labels explicitly all absorbing states of the model (for a state $s \in S$, we say that $s \models end$ iff $\forall s' \in S, a \in A \bullet \Delta(s, a)(s') \neq 0 \Leftrightarrow s = s'$).
   - r is a reward structure labeling goal states with their associated utility, computed based on the preferences defined in the utility profile. Specifically, the reward of an arbitrary state s is defined as $r(s) = \sum_{i=1}^{q} w_i \cdot u_i(v_i^s)$ if $s \models goal$ ($r(s) = 0$ if $s \not\models goal$), where u_i and $w_i \in [0, 1]$ are the utility function and weight for quality dimension $i \in \{1, \ldots, q\}$, respectively. v_i^s is the value that the state variable associated with the architectural property representing quality attribute i takes in state s.

[2]See Appendix A.2 in [6] for details.

The state space and behaviors of the game are generated by performing the alphabetized parallel composition of a set of stochastic processes under the control of the two players in the game (i.e., processes synchronize only on actions appearing in more than one process). Specifically:

- The self-adaptive system (player *sys*) controls the choices made by two different processes:

  The *controller*, which corresponds to a specification of the behavior followed by the adaptation layer of the self-adaptive system, and can trigger the execution of tactics on the target system under adaptation. The set of actions available to the controller process A_{sys} corresponds to the set of available tactics in the adaptation model. Each action $a \in A_{sys}$ is encoded in a command:[3]

  $$[a]\ C_a \wedge \neg goal \wedge t = sys \text{ -> } t' = env$$

  Where the guard includes: (i) the conjunction of architectural constraints that limit the applicability of tactic a (abstracted by C_a, e.g., a new server cannot be activated in Znn.com's pool if all of them are already active, Listing 4, line 4), (ii) a predicate $\neg goal$ to avoid expanding the state space beyond the satisfaction of the adaptation goal, and (iii) a predicate to constrain the execution of actions of player *sys* to states $s \in S_{sys}$ (control of player turns is made explicit by variable t).
  An additional command in the controller introduces absorbing states that succeed every state satisfying *goal*:

  $$[\,]\ goal \wedge \neg end \wedge t = sys \text{ -> } end' = true$$

  All the local choices regarding the execution of controller actions are specified nondeterministically in the process, since this will enable the unfolding of all the potential adaptation executions when performing the parallel composition of the different processes in the model.

  The *target system*, whose set of available actions is also A_{sys}. Action executions in the target system synchronize with those in the controller process on the same action names. In this case, each action can be encoded in one or more commands of the form:

  $$[a]\ pre_a \text{ -> } p_a^1 : post_a^1 + \cdots + p_a^n : post_a^n$$
  $$[a]\ pre'_a \text{ -> } p_a'^1 : post_a'^1 + \cdots + p_a'^n : post_a'^n \quad \ldots$$

  Hence, a specific action in the controller can synchronize with any of the alternative executions of the same action in the target system. This models the different execution instances that the same tactic can have on the target system (e.g., when the controller enlists a server in Znn.com in Listing 4, line 4, the target system can activate any of the alternative available servers, as specified in Listing 3, line 9). Each one of these commands is guarded by the precondition of a tactic's execution instance, denoted by pre_a (e.g., a specific server needs to be active to be discharged). Moreover, the execution of a command can result in a number of alternative updates on state variables (along with their assigned probabilities p_a^i) that correspond to the different probabilistic outcomes of a given tactic execution instance, denoted by $post_a^i$ (e.g., the activation of a specific server can result in a success with probability p, and fail with $1-p$).

- The environment (player *env*) controls a single process *environment*, which models potential disturbances in the execution context that are out of the system's control (e.g., network latency). The environment process is specified as a set of commands with asynchronous actions $a \in A_{env}$, and similarly to the controller process, its local choices are specified nondeterministically to obtain a rich specification of the environment's behavior. Each one of the commands follows the pattern:

  $$[a]\ C_a^e \wedge \neg end \wedge t = env \text{ -> } p_a^1 : post_a^1 \wedge t' = sys + \ldots + p_a^n : post_a^n \wedge t' = sys$$

  Where C_a^e abstracts the conjunction of environment constraints for the execution of action a (e.g., a threshold for the maximum latency that can be introduced in the network, in the case of Znn.com), and $\neg end$ prevents the generation of further states for the game. The command includes one or more updates, along with their associated probabilities. Each alternative update corresponds to one probabilistic outcome of the execution of a ($post_a^i$), and yields the turn to the system player.

2. **Strategy Synthesis.** Consists of generating a memoryless, deterministic strategy in $\mathcal{G}$ for player *sys* that has the objective of reaching a state satisfying *goal* that maximizes the value of reward r (i.e., utility). The specification for the synthesis of such a strategy is given as a rPATL property following the pattern $\langle\langle sys \rangle\rangle \mathsf{R}^r_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ end]$, which enables the quantification of the maximum accrued utility reward r along paths leading to states satisfying *end* that can be guaranteed by the system player, independently of the strategy followed by the environment player.[4]

In the remainder of this section, we illustrate our approach by describing a SMG model of Znn.com. We then show how to synthesize optimal adaptation plans wrt a given utility profile encoded as a reward structure. Finally, we present some results that indicate the scalability of the approach.

4.1 SMG Model Specification

Our formal model of Znn.com is implemented in PRISM-games [7], a probabilistic model-checker for modeling and analyzing SMGs. The game is played in turns by two players in control of the behavior of the environment and the system (i.e., controller plus target system), respectively. The SMG model consists of:

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, Listing 2). Player sys controls all transitions that belong to the controller module

[3]We illustrate our approach to modeling the SMG using the syntax of the PRISM language [16] for Markov Decision Processes (MDPs), which are encoded as commands:

$$[action]\ guard \text{ -> } p_1 : u_1 + \ldots + p_n : u_n$$

Where *guard* is a predicate over the model variables. Each update u_i describes a transition that the process can make (by executing *action*) if the guard is true. An update is specified by giving the new values of the variables, and has an assigned probability $p_i \in [0,1]$. Multiple commands with overlapping guards (and probably, including a single update of unspecified probability) introduce local nondeterminism.

[4]Note that the accrued reward along paths quantified in a path formula $\mathsf{F}\phi$ does not account for the utility in states where ϕ is satisfied. That is why in order to account for the utility reward in goal states, we need to introduce additional absorbing states succeeding goal states where *end* is satisfied (in contrast with using a formula $\langle\langle sys \rangle\rangle \mathsf{R}^r_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ goal]$).

(Listing 4), including all the transitions that synchronize with the target_system module (Listing 3), which represent the execution of tactics upon the target system (explicitly listed between square brackets in Listing 1, line 2). Global variable t in Listing 1, line 3 is used to control turns in the game and make players alternate, ensuring that for every state of the model, only one player can take action.

```
1  player env environment endplayer
2  player sys [enlist_server], [discharge_server], [raise_fidelity],[lower_fidelity],
       controller, target_system endplayer
3  const TE=0, TS=1; global t:[TE..TS] init TE;
```

Listing 1: Player definition for Znn.com's SMG.

Environment. Listing 2 shows the encoding used for a simple version of Znn.com's environment, which is able to introduce a disturbance in response time by placing an arbitrary amount of network latency in the execution context. Line 1 defines the constants that parameterize its behavior: [5]

- MIN_LATENCY and MAX_LATENCY are the minimum and maximum magnitude of the disturbance (i.e., network latency) that the environment can introduce in the system's state after the execution of a tactic.
- MAX_TOTAL_LATENCY constrains the maximum total disturbance that the environment can introduce throughout plan execution (and hence, that the plan can tolerate without requiring replanning).

```
1  const MIN_LATENCY, MAX_LATENCY, MAX_TOTAL_LATENCY;
2  module environment
3  rt_delta:[MIN_LATENCY..MAX_LATENCY] init MIN_LATENCY;
4  rt_delta_total:[0..MAX_TOTAL_LATENCY] init 0;
5    [](t=TE)&(!end)&(X+rt_delta_total<MAX_TOTAL_LATENCY)->(
         rt_delta'=X)&(rt_delta_total'=rt_delta_total+X)&(t'=TS); ...
6  endmodule
```

Listing 2: environment module.

Moreover, lines 3-4 declare the different variables that define the state of the environment: rt_delta corresponds to the additional latency introduced by the environment in the current step, and rt_delta_total keeps track of the accumulated latency introduced by the environment during the plan.

Each turn of the environment consists of setting the amount of network latency for the current step in the plan. This is achieved through a set of commands that follow the pattern shown in Listing 2, line 5: the guard in the command checks that (i) it is the turn of the environment to move, (ii) an absorbing state has not been reached yet (!end), and (iii) the value of accrued network latency has not reached its limit. If the guard is satisfied, the command: (i) sets the value of network latency for the current plan step (represented by X in the command), (ii) adds the value of network latency to the accumulator rt_delta_total, and (iii) modifies the value of t, yielding control to the system player. Note that there may be as many of these commands as possible values can be assigned to the network latency for the current plan step.

Target 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, respectively.
- MAX_RT and INIT_RT, which specify the system's maximum and initial response times, respectively.
- MAX_FIDELITY, MIN_FIDELITY, and STEP_FIDELITY, which specify the minimum and maximum fidelity levels of a server, as well as the step among fidelity levels.
- INIT_SX_ON and INIT_SX_F, specify whether server X is initially active and its initial fidelity level, respectively.
- SX_BSP is the boot success probability for server X.

```
1  formula es_f_rt=rt-1000>=0?(rt-1000<=MAX_RT?rt-1000:
       MAX_RT):0;
2  const MIN_SERVERS, MAX_SERVERS, MAX_RT, INIT_RT,
       MIN_FIDELITY, MAX_FIDELITY, STEP_FIDELITY;
3  const INIT_S1_ON, INIT_S2_ON, ...; const INIT_S1_F, INIT_S2_F, ...;
       const S1_BSP, S2_BSP, ...;
4  module target_system
5  rt:[0..MAX_RT] init INIT_RT;
6  s1_on : bool init INIT_S1_ON; s2_on : bool init INIT_S2_ON; ...
7  s1_f : [MIN_FIDELITY..MAX_FIDELITY] init INIT_S1_F;
8  s2_f : [MIN_FIDELITY..MAX_FIDELITY] init INIT_S2_F; ...
9    [enlist_server](!s1_on)->S1_BSP:(s1_on'=true)&(rt'=es_f_rt+rt_delta
         )+(1-S1_BSP):(rt'=rt+rt_delta); ...
10   [discharge_server](s1_on)->(s1_on'=false)&(rt'=ds_f_rt+rt_delta); ...
11   [raise_fidelity](s1_on)&(s1_f+STEP_FIDELITY<=MAX_FIDELITY)
         ->(s1_f'=s1_f+STEP_FIDELITY)&(rt'=rf_f_rt+rt_delta); ...
12   [lower_fidelity](s1_on)&(s1_f-STEP_FIDELITY>=MIN_FIDELITY)
         ->(s1_f'=s1_f-STEP_FIDELITY)&(rt'=lf_f_rt+rt_delta); ...
13 endmodule
```

Listing 3: target_system module.

Moreover, the module includes variables to represent the current system state: rt is the system's response time, sx_on indicates whether server x is currently active, and sx_f indicates the current fidelity level of server x.

Each tactic that can be executed upon the target system is represented by a set of commands labelled with the tactic's name. Each of these commands corresponds to a different potential execution of the same tactic (e.g., enlist_server can be executed upon any of the inactive servers in the system). Every command is guarded by the applicability condition of a specific tactic execution instance (e.g., a specific server should be inactive to be enlisted), updating the different state variables of the system according to the effect of the tactic's execution and the potential disturbances prescribed by the environment (in this case, this corresponds to network latency being factored into the system's response time):

- enlist_server activates an inactive server x with probability SX_BSP, setting the value of sx_on to true, and updating the value of response time according to the impact defined for the tactic in Table 1 (formula es_f_rt, line 1). Alternatively, server activation fails with probability 1-SX_BSP.
- discharge_server deactivates an active server, increasing response time according to formula ds_f_rt (analogous to es_f_rt, but with opposite effect).
- lower_fidelity lowers the fidelity of an active server one step, decreasing response time.
- raise_fidelity raises the fidelity level of an active server one step, increasing response time.

Controller. Module controller (Listing 4) models the behavior of the controller as a fully nondeterministic process in which each of the actions corresponds to a tactic that can be executed on the target system. The module uses an explicit representation of the goal of adaptation to discriminate goal states for the adaptation plan. In this case, we assume that the objective of adaptation is lowering response time below some threshold, and hence the formula goal (line 1) is satisfied when response time is below THRESHOLD_RT.

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

```
1  formula goal=(rt<THRESHOLD_RT?true:false); ...
2  module controller
3  end : bool init false;
4    [enlist_server] (t=TS) & (!goal) & (s<MAX_SERVERS)->(t'=TE);
5    [discharge_server](t=TS)&(!goal)&(s>MIN_SERVERS)->(t'=TE);
6    [raise_fidelity] (t=TS) & (!goal) & (s>=MIN_SERVERS) & (f<
         MAX_FIDELITY) -> (t'=TE);
7    [lower_fidelity] (t=TS) & (!goal) & (s>=MIN_SERVERS) & (f>
         MIN_FIDELITY) -> (t'=TE);
8    [] (t=TS) & (goal) & (!end) -> 1: (end'=true);
9  endmodule
```

Listing 4: controller **module.**

The controller module contains a set of synchronous commands, each one corresponding to one of the tactics that can be executed on the target system. Each one of them can synchronize with any of the commands labeled with the same action name in the target_system module (e.g., the command in Listing 4, line 5, could synchronize with the command in Listing 3, line 10 to discharge server 1, or with other commands to discharge any of the other active servers). The system player is in control of all these synchronous transitions (as defined in Listing 1, line 2). A synchronous command for the execution of a tactic can only be fired if: (i) It is the turn of the system to take action, (ii) the adaptation goal has not been satisfied, and (iii) there are no additional architectural constraints preventing the execution of the tactic (e.g., no servers can be discharged if the system is running with MIN_SERVERS active servers).

In addition, the module also contains an asynchronous command (line 8) that sets the value of the end variable to true whenever the adaptation goal is satisfied, resulting in the creation of absorbing states in the model.

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

```
1  const W_UR, W_UF, W_UC;
2  formula uR = (rt>=0 & rt<=100? 1:0)
3          +(rt>100&rt<=200?1+(−0.01)∗((rt−100)/(100)):0) ...
4          +(rt>2000&rt<=4000?0.25+(−0.25)∗((rt−2000)/(2000)):0)
5          +(rt>4000 ? 0:0); ...
6  rewards "rIU" goal : W_UR∗uR + W_UF∗uF + W_UC∗uC; endrewards
```

Listing 5: **Utility functions and preferences encoding.**

Listing 5 illustrates in lines 2-5 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 2. The formula in the example computes the utility for performance, based on the value of the variable for system response time rt. Moreover, line 6 shows how a reward structure can be defined to compute a utility value for any state by using utility preferences (defined in line 1 as weights W_UR, W_UF, and W_UC for performance, fidelity, and cost respectively). Labelling goal states in the model with utility rewards in such a way effectively enables the synthesis of optimal player strategies leading to target system configurations that maximize utility. We discuss how such strategies are synthesized in the next section.

4.2 Strategy Synthesis

To produce optimal adaptation plans, we use rPATL specifications as input to PRISM-games, which can synthesize optimal strategies for such properties, given a SMG model. Adaptation plan synthesis involves two steps:

1. Instantiating the formal model of the SMG for a specific exemplar of the system, its initial configuration, utility preferences, and adaptation goal. In the model presented in Section 4.1, this is achieved by providing values for the constants that parameterize it (e.g., available servers).
2. Generating a strategy for the instance of the model based on a rPATL property specified according to the pattern $\langle\langle \mathsf{sys} \rangle\rangle \mathsf{R}^{\mathsf{rIU}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{end}]$, which enables the synthesis of a strategy to maximize the accrued utility reward rIU.

Figure 2 shows an example of strategy synthesis to optimize different utility rewards in a configuration of Znn including 3 servers (S1, S2, and S3), each of which can serve contents either with low (1) or high (2) fidelity. While S1 and S2 are 100% reliable, S3 has a boot success probability S3_SBP=0.3. Every state node in the figure[6] is labelled with a tuple (rt, s1_on, s2_on, s3_on, s1_f, s2_f, s3_f, end) for system response time, server activity and fidelity levels, and end state information, respectively.

In its initial configuration, only S1 is active, and response time is 2000ms (state 0). The goal for adaptation in this example is lowering response time below THRESHOLD_RT (Listing 4, line 1), which in this case is set to 1500ms. All states in which the goal is satisfied are labelled with different utility values that favor the maximization of performance, fidelity, and cost respectively. These values correspond to reward structures analogous to the one in Listing 5, line 6, using the utility functions in Table 2 and weights defined in Table 3. Absorbing states are shaded in gray.

Table 3: Sample utility preferences for Znn.

Reward structure	W_UR	W_UF	W_UC
rIUR	1.0	0	0
rIUF	0	1.0	0
rIUC	0	0	1.0

The highlighted paths in the figure correspond to the different plans generated. The solid/red path corresponds to the maximization of both the performance (rIUR) and cost rewards (rIUC), leading to goal state 4 after lowering fidelity on S1 and enlisting S2. Although the value of rewards for rIUR and rIUC in goal states 4 and 5 is the same, the probability of achieving that utility in state 5 is lower, since the state is reached only with probability 0.3 through the activation of the less reliable server S3 from state 1, instead of S2 (which can fail to boot with probability 0.7).

A similar phenomenon occurs in the dashed/blue path that optimizes fidelity (rIUF), leading to state 8. Again, state 9 shows a similar value for rIUF, but strategy synthesis favors activating the most reliable server S2, instead of S3.

5. EXPERIMENTAL RESULTS

To assess the scalability of the approach, we used different instances of Znn.com with an increasing number of components, measuring execution time for plan generation, including both SMG model construction and strategy synthesis.

We obtained results for three variants of the problem:

1. Non-probabilistic. The outcome of executing an adaptation tactic upon the target system can only have one possible outcome (e.g., enlisting a specific server will always

[6]The example does not include environmental disturbance. Only system player nodes are shown for clarity.

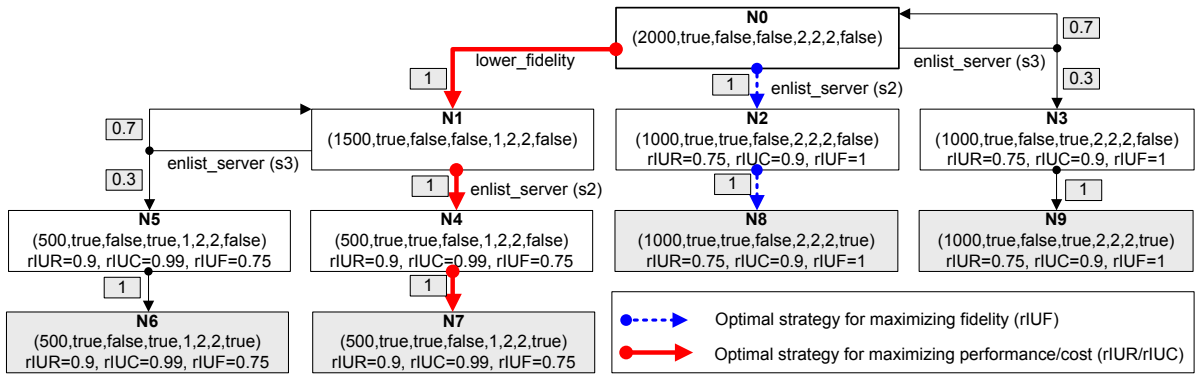

Figure 2: Adaptation plans: performance/cost (red-solid) *vs.* fidelity (blue-dashed) optimization.

result in the successful activation of the server). In the context of our model, this corresponds to setting to one all successful server activation probabilities (SX_BSP). Moreover, in this variant there are no environmental disturbances (i.e., MAX_LATENCY=0).

2. Probabilistic, in which there are different potential outcomes for the execution of some adaptation tactics, each one assigned with a probability. In our experiments, we introduce probabilistic outcomes by assigning random boot success probabilities to all servers in the system.
3. Probabilistic, including disturbance. This variant is similar to the probabilistic one, but also accounts for disturbances in response time caused by network latency. In particular, the variation in response time after the execution of every tactic corresponds to the combination of the tactic's impact on response time and the network latency introduced by the environment (up to 200ms).

Experiments were carried out by using PRISM-games beta r5753 64-bit on a machine running OS X 10.9.1, with an Intel Core 2 Duo processor and 4GB of RAM.

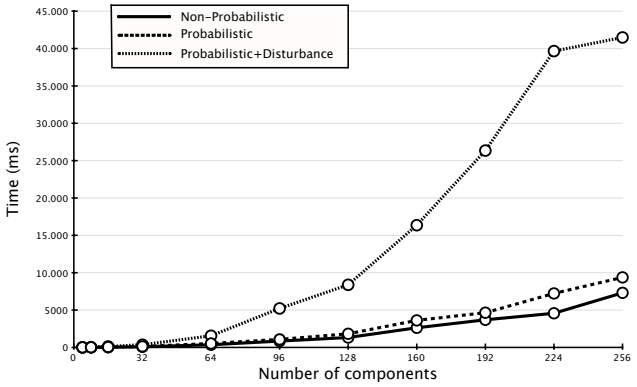

Figure 3: Execution time for plan generation.

Figure 3 shows execution times (in milliseconds) for the three variants of optimal adaptation plan generation in instances of Znn that range between 4 and 256 components.

All the variants take increasing amounts of time to produce the optimal solution as the number of components grows. In all instances of the model with up to 128 components, the two variants without disturbances (both probabilistic -dashed line- and non-probabilistic -solid line) are able to produce an optimal solution in less than 2s, whereas in all cases (up to 256 components), the optimal solution can be found in less than 10s. In general, adding probabilistic outcomes to tactics causes a marginal increment in execution time wrt the non-probabilistic variant. In contrast, the probabilistic variant that accounts for environmental disturbance (dotted line) results in remarkably higher execution times. In particular, for model instances of up to 128 components the optimal solution is obtained in less than 9s, whereas for the model with 256 components the time increases to 42s.

Note that while accounting for environmental disturbance increases the computational cost of planning, it also reduces the need to replan when the outcome of actions deviates from what is expected during execution. If the behavior of the environment does not disturb the system beyond the limits for which the solution has been built, the cost of replanning is approximately 11% of the overall time required to generate the initial plan, which is the average cost of optimal strategy synthesis (the SMG does not have to be rebuilt). In contrast, any deviation from the outcome of actions in any of the other two variants requires full replanning, including SMG construction. In general, the most appropriate choice for plan generation depends on the level of available information about the environment's behavior, which will enable to analyze in each case if the upfront cost in model generation will pay off during plan execution.

6. RELATED WORK

Different approaches in the literature tackle the challenge of planning for architecture-based self-adaptation via model checking: Tajalli et al. [19] introduce PLASMA, an approach that uses plan-based and architecture-based mechanisms for model-driven adaptation. PLASMA embodies planning via model checking as described in [13]. The approach is able to derive plans (as sets of state-action pairs) for a given goal and initial state provided by the user. However, this process does not attempt to optimize the provided solution.

Sykes et al. present in [18] two variants of an approach for the assembly of component configurations that uses non-functional information to guide the process. The first variant (aggregate selection) guarantees obtaining an optimal solution in terms of utility, but it is costly, since in the worst case it needs to generate the full list of candidate solutions to rank them. A second variant (incremental selection) is a greedy algorithm that lowers the computational cost of the process, but is unable to guarantee an optimal solution.

Although the former approaches enable the synthesis of sophisticated adaptation plans, they do not account for uncertainty and assume that replanning is necessary whenever the outcome of an action deviates from its expected effect.

In contrast, some other approaches that explicitly deal with uncertainty are more limited in the mechanisms used

to adapt the system (e.g., parameter optimization, or instantiation of predefined workflows): Calinescu and Kwiatkowska [4] introduce an autonomic architecture that uses Markov-chain analysis to dynamically adjust the parameters of an IT system according to its environment and goals. Their work assumes a set of Markov-chains describing the components of a system and uses PRISM to derive optimal parameter values to improve system operation.

Epifani et al. [10] present the KAMI methodology and framework to keep models alive by feeding them with runtime data that updates their internal parameters. The framework focuses on reliability and performance, and uses Discrete-Time Markov Chains (DTMCs) and Queuing Networks as models to reason about non-functional properties.

Calinescu et al. [3] extend different elements from [4] and [10] for defining a tool-supported framework for the development of adaptive service-based systems. QoS requirements are translated into PCTL formulas used for enforcing optimal system configurations. The approach requires designers to supply Markov chains that describe the available services, and a predefined abstract workflow upon which different candidate services will be selected to fulfill the abstract roles and obtain new system configurations.

Prior work [5] presents an analysis technique based on model checking of SMGs to quantify the potential benefits of employing different types of algorithms for self-adaptation. The approach is limited to worst case scenario analysis of the maximum utility that an optimal decision maker can guarantee when proactively adapting the system, assuming that perfect predictions of the environment's future behavior are available. In contrast, in this paper the formal model is tailored to enable the synthesis of the state space of all potential solutions for a adaptation problem, enabling the synthesis of optimal reactive adaptation plans.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented an approach for the automatic synthesis of optimal adaptation plans in architecture-based self-adaptive systems. The approach is based upon the Rainbow approach, and a technique for model checking SMGs implemented in PRISM-games. We have illustrated how to model the interaction of the system and the environment by encoding adaptation tactics that include probabilistic outcomes, utility functions and preferences, and an explicit model of the environment's behavior. Furthermore, we have assessed the scalability of optimal plan synthesis on different instances of Znn.com. Our results show that the approach scales reasonably well and is suitable for runtime plan synthesis when considering probabilistic outcomes of actions. However, in cases that also incorporate an explicit representation of the environment's behavior, the scalability of the approach is more limited in larger problem instances.

Regarding future work, we aim at exploring the combined use of predefined adaptation strategies with runtime generated plans. Hence, the adaptation manager could use a suitable predefined adaptation strategy for a given scenario, and if such strategy does not exist, generate a plan for it. Moreover, we intend to carry out a comprehensive assessment to determine if the scalability of the approach is adequate in the context of different types of systems, considering as major factors size (i.e., number of components and adaptation tactics), as well as time constraints.

8. ACKNOWLEDGEMENTS

This work is supported in part by awards N000141310401 and N000141310171 from the Office of Naval Research, CNS-0834701 from the National Science Foundation, and by the National Security Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Office of Naval Research or the U.S. government.

9. REFERENCES

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

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

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

[4] R. Calinescu and M. Z. Kwiatkowska. Using Quantitative Analysis to Implement Autonomic IT Systems. In *ICSE*, 2009.

[5] J. Cámara et al. Stochastic Game Analysis and Latency Awareness for Proactive Self-Adaptation. In *SEAMS*. ACM, 2014.

[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 *TACAS*, volume 7795 of *LNCS*. Springer, 2013.

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

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

[10] I. Epifani et al. Model Evolution by Run-Time Parameter Adaptation. In *ICSE*. IEEE CS, 2009.

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

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

[13] F. Giunchiglia et al. Planning as model checking. In *ECP*, volume 1809 of *LNAI*. Springer, 1999.

[14] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36, 2003.

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

[16] M. Kwiatkowska et al. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. of CAV'11*, volume 6806 of *LNCS*. Springer, 2011.

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

[18] D. Sykes et al. Exploiting non-functional preferences in architectural adaptation for self-managed systems. In *SAC*. ACM, 2010.

[19] H. Tajalli et al. PLASMA: a plan-based layered architecture for software model-driven adaptation. In *ASE*. ACM, 2010.