Hybrid Planning for Decision Making in Self-Adaptive Systems

Ashutosh Pandey
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
Email: ashutosp@cs.cmu.edu

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

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

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

***Abstract*—Run-time generation of adaptation plans is a powerful mechanism that helps a self-adaptive system to meet its goals in a dynamically changing environment. In the past, researchers have demonstrated successful use of various automated planning techniques to generate adaptation plans at run time. However, for a planning technique, there is often a trade-off between timeliness and optimality of the solution. For some self-adaptive systems, ideally, one would like to have a planning approach that is both quick and finds an optimal adaptation plan. To find the right balance between these conflicting requirements, this paper introduces a hybrid planning approach that combines more than one planner to obtain the benefits of each. In this paper, to instantiate a hybrid planner we combine deterministic planning with Markov Decision Process (MDP) planning to obtain the best of both worlds: deterministic planning provides plans quickly when timeliness is critical, while allowing MDP planning to generate optimal plans when the system has sufficient time to do so. We validate the hybrid planning approach using a realistic workload pattern in a simulated cloud-based self-adaptive system.**

I. Introduction

Often self-adaptive software systems operate in uncertain environments. To meet their requirements under dynamically changing environments, self-adaptive systems seek to change their behavior by determining adaptation strategies at run time.

Researchers have proposed various approaches to accomplish this. Systems such as Rainbow have a repertoire of predefined adaptation strategies created at design time by domain experts based on their past troubleshooting experience [5]. When an adaptation is triggered at run time, the repair strategy that maximizes the expected utility is selected from the set of predefined strategies. In the artificial intelligence community, this approach fits in the category of case-based reasoning (CBR), which solves new problems based on the solutions to similar problems [12]. Since strategies are not generated at run time and the calculation of expected utility is fast (by virtue of the design of the repair strategy language), the process of selecting an adaptation strategy is quick. However, optimality of repair (in a utility-theoretic sense) cannot be guaranteed, since the set of predefined strategies may not be sufficient to handle unforseen problems or environments.

In contrast to having a predefined set of strategies, approaches have been suggested to generate adaptation plans at run time. In particular, researchers have demonstrated the potential of various automated planning techniques for run-time plan generation in the context of self-adaptive software systems [10][27]. Moreover, various frameworks have been suggested to support run-time plan generation using automated planning [6][4]. Since automated planning explores a large space of possible repair strategies that can be used in the current state of the system and its environment, it is often able to provide nearly optimal solutions or find repairs for unexpected situations.

Numerous automated planning techniques have been developed by the artificial intelligence community providing various tradeoffs in planning time versus quality [21]. For example, planners that incorporate uncertainty -- an inherent feature of most adaptive systems -- often have exponentially larger state spaces to explore compared with deterministic planners. But the resulting plans are typically much better. Therefore, developers of self-adaptive systems are typically faced with a dilemma: whether to use fast but sub-optimal planning approaches (such as case-based reasoning or deterministic planning) vs. slow but nearly optimal planning approaches that deal with uncertainty (such as Markov Decision Process [20] or Partially Observable Markov Decision Processes [22]).

For many systems this choice is problematic: they need both quick actions under some circumstances, but can tolerate more deliberative planning to improve system performance over the long term. Consider, for example, a cloud system such as Amazon Web Services, where as per their service level agreement (SLA), it is critical to maintain an up-time percentage of at least 99.95% in any monthly billing cycle.[1] For such systems, in case of a failure, rapid response is needed to maintain this availability guarantee. However, for such a system, there are typically other quality concerns such as minimizing operating cost. In other words, the overall utility of the system is dominated by the key quality constraints, but other concerns remain relevant. Netflix is another example of such a system, where managing the overall latency of response to clients is critical to good user experience, in spite of the desire to minimize resource usage, and thus to bring down operating cost.[2]

Ideally, for such systems, a planning approach that can find optimal adaptation strategies quickly is needed. However, it is hard to find a single automated planning approach that satisfies both of these conflicting requirements. To deal with that trade-off, this paper introduces a novel *hybrid planning* approach that combines multiple automated planning technologies to bring their benefits together. The hybrid planning approach provides a quick response in case of a time-critical problem, such as an SLA constraint violation. However, when there is sufficient time, it can generate a close-to-optimal plan to maximize the overall long-term utility of the system. This idea of hybrid planning is

[1] https://aws.amazon.com/ec2/sla/

[2] http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html

akin to human decision making: depending upon factors such as available planning time, humans apply different levels of deliberation while making real life decisions [38].

Specifically, to instantiate a hybrid planner, we combine a fast, deterministic planning approach with a slow, but optimal, MDP planning approach.[3] In case of a constraint violation, deterministic planning is triggered to provide a rapid response. This planner is able to reduce the planning time by ignoring uncertainty in the planning domain, thereby reducing the state space to be explored.[4] However, ignoring uncertainty adversely impacts the optimality of plans. To deal with this issue of sub-optimal plans, while the deterministic plan is being executed, MDP planning is performed in the background to generate an optimal adaptation policy by taking uncertainty into account.

However, the combination of deterministic planning and MDP planning can only be effective if the MDP policy is able to seamlessly take over the plan execution from the deterministic plan. Fortunately, the structure of an MDP policy increases the chance of seamless transfer of plan execution from the deterministic plan to the MDP policy. If the deterministic and the MDP planning have the same initial state, once the MDP policy is ready, it takes over the plan execution from the deterministic plan in any of the future states, ensuring the optimal policy execution thereafter (detailed explanation in Section IV). However even if the MDP policy is optimal, since it takes over execution from a (possibly) sub-optimal plan, the combined plan might be sub-optimal.

As an initial feasibility study for the hybrid planning approach, we conducted experiments in a simulated cloud-based self-adaptive setting using a realistic workload pattern from the WorldCup '98 website [16]. This setting deals with two kinds of uncertainties: uncertainty in the external environment (i.e., uncertain client request arrival rate), and the system (i.e., random hardware failures). In this context, the experiments demonstrate that the hybrid planning approach performs better than either deterministic or MDP planning used in isolation.

In recent work, we explored optimal planning under uncertainty by employing models able to capture probabilistic, as well as adversarial behavior of a system's environment [27]. Moreover, Moreno et al. [15] explicitly incorporate the notion of tactic latency in the execution of adaptation, yielding closer-to-optimal results, compared to latency-agnostic decision-making. In contrast, the present proposal explores the combined use of different planning techniques to explicitly deal with latency in optimal decision-making in planning for adaptation, helping to mitigate the sub-optimality that might arise from failing to make adaptation decisions in a timely manner.

The rest of the paper is organized as follows: in Section II, we introduce a motivating example that will be used throughout the paper. Section III provides background on AI planning. In Section IV, we describe the hybrid planning approach in detail, including the planning models and how the adaptation decisions are made. The Section V presents the evaluation of the hybrid approach. Section VI discusses related work in the field of automated planning and self-adaptive software systems. Finally, Section VII concludes with a discussion of future work.

[3]In general, MDP planning finds an optimal plan; however, the optimality of a plan depends on various factors such as the algorithm used and the time spent solving the MDP.

[4]Although deterministic planning can take non-negligable time for complex state spaces, it is typically orders of magnitude faster than planning under uncertainty.

II. Motivating Example

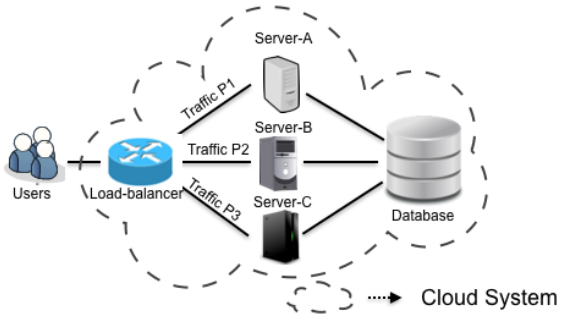

Fig. 1. High-level view for the cloud-based system

Consider we have a cloud-based self-adaptive system, as shown in Fig. 1, with a typical N-tiered architecture: a presentation tier, an application tier, and a database tier. Using the presentation tier, a client sends a request to the application tier, which interacts with the database tier to process the request. We assume the system has different types of server with varying capacity (i.e., ability to handle requests per second) and cost. Not surprisingly, the operating cost of a server increases with increase in capacity. The workload on the system depends on the request arrival rate, which is uncertain as it depends on external demand. Moreover, we assume that there is uncertainty in the system itself because of the possibility of random server crashes, which adversely impact the load bearing ability of the system.

The system needs to improve profitability by minimizing operating cost and maximizing revenue. To achieve these objectives, the system has various adaptation tactics. The system has an adaptation tactic *removeServer*, which is used to bring down the operating cost by deactivating an extra server.

Since higher perceived user response time results in revenue loss [28][31], it is desirable to maintain the response time for user requests below some threshold, say T. To accomplish this, the system can either add more servers (using the $addServer$ tactic) or control optional content such as advertisements and product recommendations.

Depending on their request, the client is presented with some mandatory content along with some optional content, which helps generating extra revenue [18]. Even though the optional content generates revenue, it uses additional computation power and network bandwidth that increase response time. A brownout mechanism is implemented to control the optional content through a dimmer variable [19]. The value of the variable is in the range [0..1], which represents the probability of including the optional content in a response. The system has tactics, $increaseDimmer$ and $decreaseDimmer$, to increase and decrease the value of dimmer variable respectively. In case of a high workload, the number of responses having optional content could be reduced by decreasing the dimmer value; however, it would also bring down the revenue generated through advertisements.

We assume there is a penalty, say P, for each request having a response time above the threshold. Therefore, in case of an average response time above the threshold, the system needs a rapid response, either by adding servers or decreasing the dimmer value. However, once the response time is under control,

the system should execute other adaptation tactics to bring down operating cost (i.e., maximize overall long-term utility).

Since the system has servers of different capacity, a round-robin strategy for assigning client requests to active servers would not be efficient. The system has another tactic *divert_traffic*, which helps the load-balancer in distributing the workload among active servers. The system uses queueing theory to determine an optimal distribution of workload [35]. Intuitively, the percentage of client requests assigned to each active server depends on their capacity.

The goal of the system is to maximize aggregate utility over a fixed window of time. The utility increases with the increase in revenue and decreases with higher operating cost. If the system runs for duration L, the utility function would be defined as:

$$U = R_O x_O + R_M x_M - P x_T - \sum_{i=1}^{n} C_i \int_0^L s_i(t)dt \quad (1)$$

where R_O and R_M is revenue generated by a response with optional and mandatory content respectively; P is the penalty for request having response time above the threshold; x_O, x_M, and x_T are number of requests with optional content, mandatory content, and having response time above the threshold, respectively; C_i is the cost of server type i, and s_i is the number of active servers of type i; n is number of different types of server.

III. Background

A. Automated Planning

Automated planning is a well established branch of artificial intelligence (AI). In essence, planning is the task of coming up with a sequence of actions to achieve a goal state from the initial state. Formally, given a set of states S, a set of actions $A : S \to S$, an initial state $s_I \in S$, and a set of goal states $S_g \in S$, the *planning problem* is the task of finding a sequence of actions that, when applied to s_I, yield goal states in S_g.

Over the years, AI researchers have developed various automated planning approaches [21]. These automated planning approaches can be broadly characterized into two categories: deterministic planning, and planning under uncertainty.

1) Deterministic Planning: Deterministic planning is applicable to domains having no uncertainty, either in action outcomes or observations of the underlying states. The simplest (although computationally expensive) way to find a deterministic plan is a brute-force search of the planning space. To speed up the process of state-space search, various algorithms [30][32] and heuristics [3][2] have been proposed by the AI community.

Since there is no uncertainty in deterministic planning, when an action a is applied to an arbitrary state s, it results in a single transition to a state s'. Therefore, deterministic planning approaches, in general, produces a linear plan, which represents a sequence of actions to get from the initial state to a goal state.

2) Planning Under Uncertainty: Researchers have developed various approaches to deal with different kinds of uncertainties in a planning domain. For instance, MDP planning deals with uncertainty in action outcomes, and POMDP planning deals with uncertainty, both in action outcomes and observations of the underlying states. By taking into consideration the various possible outcomes of actions (and system states) these approaches often provide much better plans than those rendered through a deterministic approach. However, the size of the state space grows exponentially with increasing levels of uncertainty, thereby increasing the planning time.

As we detail later, our experiments, specifically, we use finite discrete-time MDP planning as a slow, but optimal, planning approach since it (1) helps in dealing with uncertainty in the request arrival rate, and (2) generates an optimal strategy that maximizes a utility function over a time period.

Definition 3.1 (Markov Decision Process): A Markov Decision Process (MDP) is a tuple $\mathcal{M} = \langle S, s_I, A, \Delta, r \rangle$, where $S \neq \emptyset$ is a finite set of states; $s_I \in S$ is an initial state; $A \neq \emptyset$ is a finite set of actions; $\Delta : S \times A \to \mathcal{D}(S)$ is a (partial) probabilistic transition function; and $r : S \to \mathbb{Q}_{\geq 0}$ is a reward structure mapping each state to a non-negative rational reward. $\mathcal{D}(X)$ denotes the set of discrete probability distributions over finite set X.

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

We can reason about the behavior of MDPs using *policies*. A policy resolves the nondeterministic choices of an MDP, specifying which action to take in every state.

Definition 3.2 (Policy): A policy of an MDP is a function $\sigma : (SA)^* S \to \mathcal{D}(A)$ *s.t.*, for each path $\pi \cdot s$, it selects a probability distribution $\sigma(\pi \cdot s)$ over $A(s)$.

In this paper, we use policies that are *memoryless* (i.e., based solely on information about the current state) and *deterministic* ($\sigma(s)$ is a Kronecker function such that $\sigma(s)(a) = 1$ if action a is selected, and 0 otherwise). Hence, mapping states to actions based on σ is straightforward. In this paper, we obviate σ for conciseness, and refer to policies simply as maps from states to actions. The optimal policy for a MDP is one that maximizes the expected utility over all the possible policies for that MDP.

IV. Approach

The key idea behind hybrid planning is to combine the advantages of different planning approaches. For instance, to generate adaptation plans for the cloud-based self-adaptive system described in Section II, we combine a fast planning approach with an optimal, but slow planning approach. This hybrid planning uses a fast planner to handle an immediate problem, but simultaneously uses a slow planner to provide an optimal solution. For such systems, intuitively, using only the slow planner alone would increase the delay in decision making, thereby resulting in lower utility particularly when a quick response is needed; however, using the fast planner alone would provide a quick response but it would not be suitable for maximizing utility in the long term.

However, there are several research challenges in combining the multiple planning approaches: (i) identifying the set of planning approaches that could be used to find an appropriate balance between conflicting (timing versus quality) requirements, (ii) identifying planning approaches that are compatible to be used together, and (iii) determining the criteria for knowing when to invoke each of the constituent planners.

In the context of our cloud-based system, we handled these challenges by combining deterministic planning with MDP

planning. When a rapid response is required, the deterministic planner helps in quick decision making, while the MDP planner finds an optimal plan that maximizes overall utility in the long run. Therefore, the two planning approaches provide a balance between the conflicting requirements of quick and optimal planning. Moreover, as explained in the next section, the structure of MDP policy helps in the coordination between the two planning approaches.

In our cloud-based self-adaptive system, the rationale for combining the two planning approaches is the following: In case of a response time above the threshold T, deterministic planning provides a rapid response, minimizing the penalty for a high response time. Deterministic planning ignores uncertainty in the request arrival rate, which reduces the state space to be searched, thereby reducing the planning time compared to MDP planning. However, while the deterministic plan is under execution, MDP planning is performed in the background to find a better policy by taking environmental uncertainty into account.

To account for uncertainty in the MDP planning, we create an environment model using future values of interarrival time (the inverse of average request arrival rate) between two consecutive requests. When MDP planning is triggered, a time-series predictor is used to anticipate the values of interarrival time (incorporating uncertainty in the predictions). These predicted values are used to model the environment as a Markov decision process, where each possible interarrival rate is considered as an outcome of a probabilistic action made by the environment.

A. Synchronization Between Deterministic and MDP Planning

The success of the hybrid planning approach depends on the seamless transition of execution from a deterministic plan to an MDP policy. On applying an action a from the deterministic plan in state s, if the resultant state is not found in the MDP policy, the hybrid planning approach would fail to execute the optimal MDP policy for the subsequent states. However, the structure of an MDP policy increases the chance of seamless transition of execution from a deterministic plan to an MDP policy.

Fig. 2 explains the transition of execution from a deterministic plan to an MDP policy. Suppose, at state s, there is a constraint violation (e.g., response time goes above the threshold) that cannot be handled by the existing MDP policy.[5] To deal with the problem, as explained later in Algorithm 1, both deterministic planning and MDP planning are triggered simultaneously at time t_0. Assuming that deterministic planning is instantaneous, suppose it suggests an action a to be executed at time t_0. Meanwhile, MDP planning takes predicted, but uncertain, values of interarrival time into consideration and comes up with a policy, suppose at time t_1. On executing the action a, due to uncertainty in the client request arrival rate, the system could reach one of the several possible outcome states, such as $s1$, $s2$, and $s3$. If the predicted values for the interarrival time (used for MDP planning) are correct, these states will be found in the MDP policy, because the policy contains the state-action pair for all the reachable states from the state s. Therefore, the MDP policy can take over the plan execution from any of these resulting states. Moreover, due to the memoryless nature of the MDP policy, optimality of the action prescribed by the MDP policy for states such as $s1$, $s2$, and $s3$, depends only on that state, and not on any of previous states.

[5]Here a state consists of system state and environment state.

However, if the time-series predictor fails to predict the interarrival time correctly, the resulting states, such as $s1$, $s2$, and $s3$, might not exist in the MDP policy; therefore, transition from the deterministic plans to the MDP policy would not be possible. In such a situation, planning process is triggered again as explained later in the $hybrid\_planning$ algorithm.

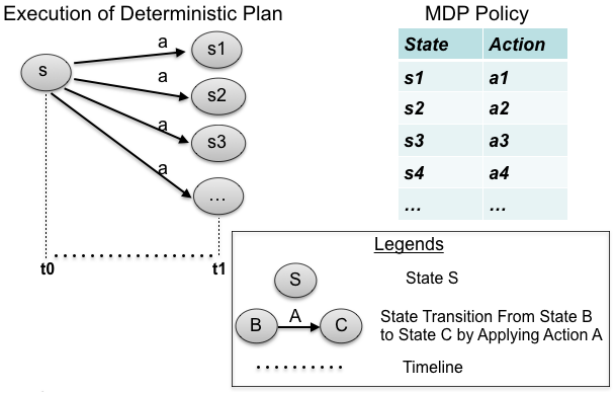

Fig. 2. Transition from Deterministic Plan to MDP policy

B. MAPE-K Loop

To bring self-adaptive ability to our cloud-based system, we implemented a MAPE-K loop [17], which has a knowledge base and three components: monitoring, analysis-planning, and execution. Even though a typical MAPE-K loop has a distinct analysis and planning component, in our implementation we merged them into a single component. This is done because in our MAPE-K loop, the planning process also analyzes if there is a need for adaptation to improve the utility.

For the running software system, the knowledge base maintains the architectural model of the target/managed system [33], which includes information such as architectural configuration of the system, quality attributes, constraints, number of active/inactive servers, current dimmer settings, work-load shared among different active servers, and average response time perceived by the clients. Moreover, the knowledge base keeps the execution status of the tactic execution since some of the tactics, such as adding a server, have execution time greater than the control loop cycle. Furthermore, the knowledge base incorporates an environment model, which includes information about the current and future (predicted) request arrival rates. All this information in the knowledge base is used to make adaptation decisions.

The monitoring component gathers information from the running system and the environment to update the architectural and the environment models. The analysis-planning component gathers information from the knowledge base periodically, with the evaluation period τ, to decide if there is a need for adaptation, defined as an opportunity to improve the utility.

To determine which adaptation tactic(s) should be executed, the analysis-planning component implements the $hybrid\_planning$ algorithm discussed in the next section. If there is an opportunity for utility improvement, the algorithm returns a non-empty set of tactics, which are passed to the execution component.

C. Hybrid Planning

At the beginning of each evaluation cycle, the analysis-planning component invokes the function $hybrid\_planning$,

listed as Algorithm 1. The inputs to the $hybrid\_planning$ function are: state S_{curr}, which includes the state of the system and the environment; and the existing MDP policy π.

Algorithm 1 hybrid_planning function returns the list of adaptation tactic(s)

Require: S_{curr}, π
1: List $tactics$
2:
3: **if** π = *null* **or** π.find(S_{curr}) = *false* **then**
4: Thread $thread \leftarrow new$ Thread()
5: $thread$.run(MDP_planning(S_{curr}))
6:
7: **if** any_constraint_violated(S_{curr}) **then**
8: $tactics \leftarrow$ deterministic_planning(S_{curr})
9: **else**
10: $tactics \leftarrow \pi$.get_tactics_from_policy(S_{curr})
11:
12: **return** $tactics$

To find an adaptation tactic corresponding to the state S_{curr}, the $hybrid\_planning$ algorithm first refers to the existing MDP policy (Algorithm 1: line 3). If S_{curr} is found in the policy, the function $get\_tactics\_from\_policy$ returns the list of tactics corresponding to all the states, starting from S_{curr}, within the evaluation cycle. Since our MDP specification and the system support concurrent execution of non-conflicting tactics,[6] the tactics in the list are triggered simultaneously at the beginning of that evaluation cycle [15]. However, if there is no opportunity to improve utility then the function $get\_tactics\_from\_policy$ returns an empty list. If the MDP policy does not exist or S_{curr} is not found in the policy then the planning process is triggered (Algorithm 1: lines 4-8).

To start planning, the $hybrid\_planning$ algorithm assigns an MDP planning process to a separate thread (Algorithm 1: lines 4-5). However, to provide a quick response in case of a constraint violation, the deterministic planning process is triggered concurrently (Algorithm 1: lines 7-8) in the main thread. With an assumption that deterministic planning is quicker than MDP planning, the function $deterministic\_planning$ would return adaptation tactics before the MDP policy is ready. Similar to our MDP planning specification, deterministic planning specification supports concurrent execution of non-conflicting tactics. Therefore, the function $deterministic\_planning$ returns list of adaptation tactics to be executed at S_{curr}. However, once the MDP policy is ready, it will take precedence over the deterministic plan.

We designed our experiments, such that MDP planning time is less than the evaluation cycle, τ. This implies that once MDP planning triggered, the policy will be ready by the beginning of next evaluation cycle. However, as explained later (in Subsection V-C), this simplifying assumption does not effect the generality of the $hybrid\_planning$ algorithm.

1) MDP Planning: Probabilistic model checking of formalisms that support the specification of nondeterminism, like MDPs, typically support synthesizing policies able to optimize an objective expressed as a quantitative property in a probabilistic temporal logic formula by resolving nondeterminism. In our case, we use the PRISM model-checker as an MDP planning tool [27]. Our PRISM specification is based on the work done by Moreno et. al., which supports concurrent tactic execution, and accounts for the uncertainty in the environment by modeling it as a probability tree [15]. Fig. 3 shows each tactic as a module, the PRISM construct for modeling concurrent processes.

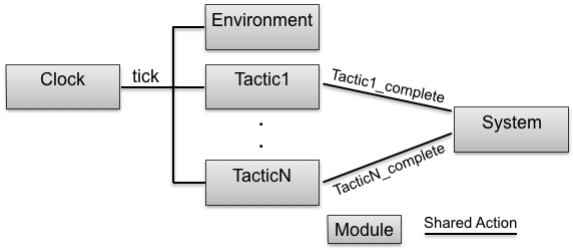

Fig. 3. Module composition of MDP specification

The execution of the MDP process is done at the granularity of evaluation period, τ. At time 0, which is also the beginning of the look-ahead horizon, the system analyzes if there is room to improve utility through adaptation. Once the system has either adapted or passed on (if no further improvement in utility of possible), the time advances τ time units -- the length of an evaluation cycle. Then, the environment takes a probabilistic transition depending on the predicted interarrival time. After that, the utility for the system, during the last evaluation cycle, is calculated and accumulated.

The clk module, shown in Listing 1, controls the passing of time and synchronizes between different modules or processes in the MDP specification. The clk module has two actions, $tick$ and $tack$, both executed once in a single evaluation cycle. The action $tick$, which advances the time, is a shared action between the environment and the tactic modules. Therefore, when action $tick$ is executed in the clk module, it is executed synchronously in all the modules sharing the action $tick$. However, the clk module cannot execute the action $tick$ until all the modules sharing the $tick$ are ready to do so. After $tick$, clk takes the action $tack$, which is also shared with the reward structure $util$ (lines 9-11). This implies that when $tack$ is executed the utility for the last period is calculated and accumulated to get the aggregate utility.

At the beginning of each evaluation cycle, the knowledge base is updated with the average interarrival rate for the previous period. When MDP planning is triggered, using the past observations, the time-series predictor estimates the future

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

Listing 1. Clock module and reward structure

[6]The tactics $addServer$ and $removeServer$ are conflicting, since they cancel each other's effects. Similarly, $increseDimmer$ and $decreaseDimmer$ is another pair of tactics that is conflicting in nature.

```
module env
s : [0..N−1] init 0;

[tick] s = 0 −> 0.185 : (s' = 1) + 0.63 : (s' = 2) + 0.185 : (s' = 3);
[tick] s = 1 −> 0.185 : (s' = 4) + 0.63 : (s' = 5) + 0.185 : (s' = 6);
...
endmodule

formula stateValue = (s = 0 ? E_0 : 0) +
                     (s = 1 ? P5_E_1 : 0) +
                     (s = 2 ? P50_E_1 : 0) +
                     ...
```

Listing 2. Environment module in PRISM

```
module sys
    ...
    [addServer_complete] servers < MAX_SERVERS −> 1 : (servers' =
        servers + 1);
    [removeServer_complete] servers > 1 −> 1 : (servers' = servers − 1);
    [increaseDimmer_complete] dimmer < DIMMER_LEVELS −> 1 :
        (dimmer' = dimmer + 1);
    [decreaseDimmer_complete] dimmer > 1 −> 1 : (dimmer' = dimmer −
        1);

    // Percentage of traffic is discretized to the values 0, 25, 50, 75, and 100.
    // Assign 25% traffic to server Type−A, 25% to server Type−B, and
        50% traffic to server type−C,
    [divert_25_25_50] active_servers_A > 0 & active_servers_B > 0 \
        & active_servers_C > 0 −> 1 :\
            (traffic_A' = 1) & (traffic_B' = 1) & (traffic_C' = 2);
    ...
endmodule
```

Listing 3. System module

interarrival time for the requests. As we predict farther into the future, the accuracy of prediction decreases, which could lead to a sub-optimal MDP policy; moreover, it also increases the state space leading to longer planning time. Therefore, MDP planning is done for a finite horizon, $lookAheadHorizon$. This implies that when MDP planning is triggered using predicted values for the request arrival rate to $lookAheadHorizon$ evaluation cycles, PRISM generates an MDP policy for that period. This period of $lookAheadHorizon$ evaluation cycles (i.e., $lookAheadHorizon \times tau$ time units) is known as the MDP planning horizon.

The environment module, env, is shown in Listing 2. To build a model for the environment, we build a probability tree that represents both the predicted interarrival rates and the uncertainty of the request arrival rate. We use Extended Pearson-Tukey (EP-T) three-point approximation [34] that consists of three points that correspond to the 5th, 50th, and 95th percentiles of the estimation distribution, with probabilities 0.185, 0.630, and 0.185, respectively.

The $system$ module (Listing 3), shares tactic completion actions with each tactic module. This implies that when a tactic execution is completed, the corresponding shared action is executed synchronously in the $system$ module, which updates the system configuration accordingly. In other words, the $system$ module represents the knowledge-base, which keeps track of the current configuration of the system such as the number of active servers, the value of the dimmer variable, and the workload assigned to each server. This information is used to calculate utility. The $system$ module uses queueing theory to estimate the future average response time based on request arrival rate and system configuration. We use a variation of a $M/G/1/PS$ queueing model [35] that supports different capacity servers operating in parallel.

Listing 3 shows a module representing one of the tactic module, $increaseDimmer$. At the beginning of an evaluation cycle, the $increaseDimmer$ module has three choices. If the

```
module increaseDimmer
    increaseDimmer_go : bool init true;
    increaseDimmer_used : bool init false;

    [increaseDimmer_start] sys_go & increaseDimmer_go
        & increase_dimmer_applicable // applicability conditions
        −> (increaseDimmer_go' = false) & (increaseDimmer_used' =
            true);

    // tactic applicable but not used
    [pass_inc_dimmer] sys_go & increaseDimmer_go // can go
        −> (increaseDimmer_go' = false);

    [tick] !increaseDimmer_go −> 1 : (increaseDimmer_go' = true) &
        (increaseDimmer_used' = false);
endmodule
```

Listing 4. System module

tactic is applicable then module can either choose to take the action $increaseDimmer\_start$, or pass on using the action $pass\_inc\_dimmer$. However, if the tactic is not applicable e.g., dimmer variable is already at its maximum value, then tactic module executes the $tick$ action synchronously with all the modules sharing the $tick$ action. Unlike $increaseDimmer$, the $addServer$ tactic is not instantaneous. Therefore, the module representing $addServer$ has actions that represent intermediate stages while booting up a server [15].

Since MDP planning is slow, once a policy is generated, it is stored for successive evaluation cycles within the MDP planning horizon. If the system and the environment behave as expected, the policy will be applicable at the beginning of each evaluation cycle within the planning horizon. However, the policy needs to be regenerated in either of two cases: the planning horizon is over, or there is an unexpected change in environment leading to an unanticipated state not covered by the MDP policy.

At the beginning of each evaluation cycle, the current state is searched in the policy to determine which adaptation tactics need to be executed (Algorithm 1: line 3). As mentioned earlier, MDP planning is done based on predicted average interarrival time between two consecutive requests; not the actual values. Since the prediction discretizes the values, it is very likely that the actual is not one of the discrete values. In such cases the current state would not be found in the existing MDP policy.

To deal with this problem, we use a heuristic to match the current state to a state in the MDP policy. The current state is matched to a state that meets three criteria: (1) all state variables (except average interarrival time) have same value; (2) among the states meeting criterion 1, the value of average interarrival time for the matched state should be closest to the current average interarrival time; and (3) once the first two criteria are met, the difference between the values of the average interarrival time in the current state and the matched state should be within some predefined threshold of the current average interarrival time.

2) Deterministic Planning: In our implementation, we also use PRISM for the deterministic planning. Since the MDP specification has uncertainty only in the form of environmental uncertainty, to transform the MDP specification into a deterministic specification, we ignore the tree-structured prediction model for the environment. Instead, we assume that interarrival time will remain fixed at the current value. Since deterministic planning is done based on the imprecise model of the environment, the plan generated by the deterministic planning is executed only for the current evaluation cycle; the rest of the plan is ignored.

Ignoring uncertainty not only transforms the MDP specification into a deterministic specification, but also reduces the planning state space significantly resulting in a reduction of

the planning time. For a particular instance in our experiments, the MDP planning state space of more than 2 million states was reduced to about 23,000 simply by ignoring environmental uncertainty; the planning time was reduced from (approximately) 40 seconds to less than a second.

V. Evaluation

To evaluate the hybrid planning approach, we developed a simulation model of cloud-based self-adaptive system using the discrete event simulator OMNeT++.[7] The MAPE-K loop components (i.e., the knowledge base, the monitoring, the analysis-planning, and the execution) are implemented in the simulator. Moreover, in the simulator we also implemented components such as the load-balancer, and different types of servers. Besides the fundamental functionality, the components also have the logic to support various tactics at run time such as adding a server, changing dimmer settings, and load redistribution among various active servers.

A. Experiment Setup

For the experiments, we have one server of each type: type-A, type-B, and type-C. The operating cost per minute for three types of server, is \$0.5, \$0.7, and \$1.0 respectively. Intuitively, the capacity of servers, in terms of the ability to handle the client requests, increases with the cost of server. Therefore, we assume server type-C has the ability to handle 150 requests/sec when serving optional content and 300 requests/sec without serving optional content; server type-B can handle 130 requests/sec when serving optional content and 200 requests/sec without serving optional content; and server type-A can handle 50 requests/sec when serving optional content and 150 requests/sec without serving optional content. However, the service time for the request is not fixed but assumed to be normally distributed. At the beginning of the experiment, we start with a server type-A as the only active server (i.e., connected to the load-balancer to handle client requests).

We assume that the cost of a server can be covered by the revenue of handling 1/10 of its maximum capacity with optional content and the revenue of handling 2/3 of its maximum capacity without optional content. If server cost per minute is C, capacity with optional content is c_O and without optional content is c_M, then revenue for a server with optional content would be $R_O = \frac{10}{c_O}C$, and without optional content would be $R_M = \frac{3/2}{c_M}C$. For each request having a response time above the threshold value of 1 second, there is a penalty (i.e., the value of P in the Eq. 1) of -3 units per request. The value for the parameter w, used to find a matching state in an MDP policy, is set at 10%.

For the experiments, there are three dimmer levels and the experiment starts with the dimmer value 1 (i.e., all the requests are served with optional content). The evaluation period, τ, is configured as 1 minute. We assume a fixed server boot-up time of 2 minutes (2 evaluation cycles). Since at a given point, we would have a maximum of 2 inactive servers and each server has 2 minutes of boot-up time, our planning horizon for the MDP planning is 5 minutes. This heuristic gives a planning horizon long enough to go from 1 active servers to 3 active servers plus 1 additional evaluation cycle to observe the resultant utility.

To simulate realistic request arrival patterns, we used a request arrival trace (shown in Fig. 4) from the WorldCup '98 website

[7]https://omnetpp.org/

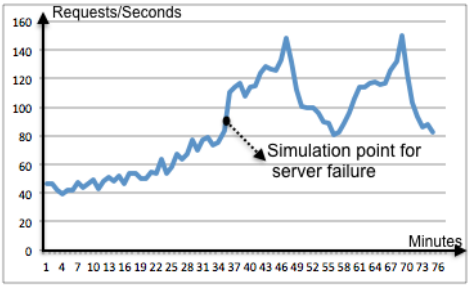

Fig. 4. Request arrival trace used as input for the experimental setup

[16]. We scaled the trace in such a way that it does not exceed the maximum capacity of the setup [15]. This trace has considerable load increases around the two games that were played on the same day. This trace contains time-stamps representing interarrival rate between the two client requests. To simulate the clients, we simulate the arrival of requests at the system with the interarrival rate obtained from the trace. Fig. 4 also highlights the point on the trace that we used to simulate a sudden crash of server type-C. The first 15 minutes of the trace are used to train the time-series predictor.

We conducted the experiments on a Ubuntu 14.04 virtual machine having 8GB RAM and 3 processors at 2.5 GHz. The state space for MDP planning approximately varies between 1.7 million and 2.2 million states and the number of transitions vary between 4.3 million to 5.8 million states. The MDP planning time varies between 35 seconds to 45 seconds. The memory for storing an MDP policy varies between 140 megabytes to 180 megabytes. In contrast, state space for the deterministic planning is less than 50,000 and the number of transitions less than 100,000. The deterministic planning time is negligible in our experiments (less than a second).

B. Results

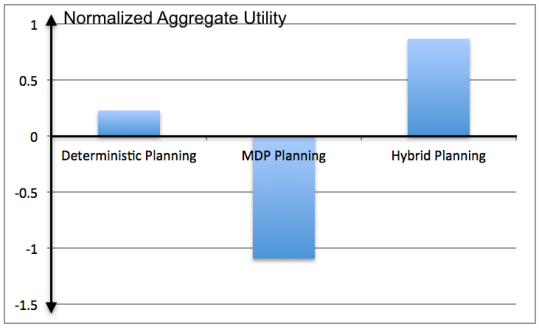

Fig. 5. Normalized aggregate utility for different approaches

To evaluate the hybrid planning approach, we compare its performance with deterministic and MDP planning used in isolation. Looking at aggregate utility, which is the sum of utility accumulated at the end of each experiment run, the hybrid planning approach performs much better compared to the other two approaches for the given request arrival trace. Fig. 5 shows the normalized aggregate utility for the three approaches. We explain the better performance of the hybrid planning approach using two scenarios: unexpected increase in request arrival rate and random server crash.

1) Unexpected Increase in Request Arrival Rate: Fig. 6, which represents the adaptation decisions made during an MDP

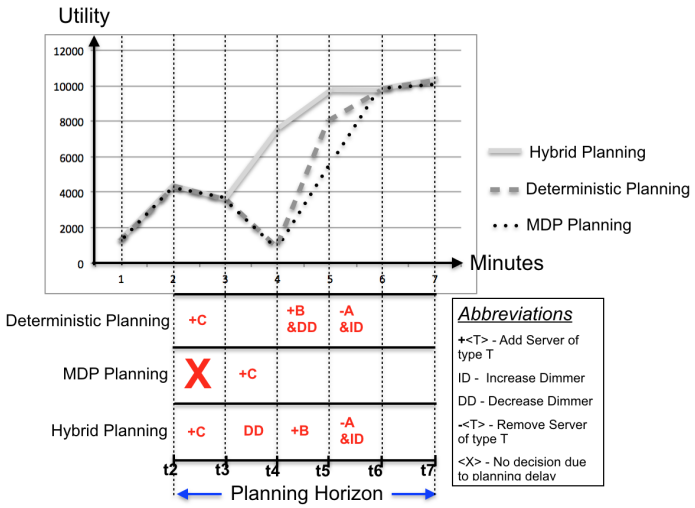

Fig. 6. Adaptation decisions made by different approaches and its impact on utility

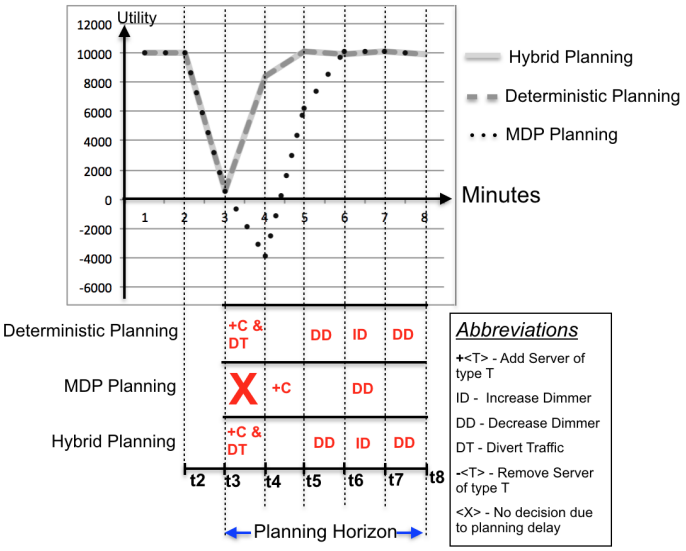

Fig. 7. Adaptation decisions made by different approaches on sudden server crash and its impact on utility

planning horizon, explains the better performance of the hybrid planning in the case of an unexpected increase in the request arrival rate. Both the evaluation cycle and the MDP planning cycle begin at time step t_2.

When deterministic planning is used in isolation, since the planning process is instantaneous it is invoked at the beginning of each evaluation cycle. In the experiment, the average response time measured at time t_2 is higher than the threshold of 1 second. Assuming the request arrival rate remains constant at the current value, deterministic planning quickly suggests an adaptation tactic to add a server of type-C at t_2. Since the server boot-up time is 2 minutes, it would be active at time t_4, and hence the effect of adding a server results in higher utility at time t_5. However, response time remains high during the period from t_2 to t_4 resulting in lower utility for that period.

In case of the slow approach (i.e., using MDP alone), when planning is triggered at time step t_2, due to planning delay it takes about one evaluation cycle to generate an MDP policy. Therefore, no adaptation decision is made at t_2 resulting in low utility for the next few evaluation cycles. However, when the MDP policy is available at time t_3, the policy suggests to add a server of type-C at t_3. This server becomes active at the time t_5 resulting in higher utility at time t_6.

In case of the hybrid planning approach, both deterministic and MDP are invoked at time step t_2. The deterministic planning suggests the tactic to add a server of type-C. However, from t_3 onwards, hybrid planning follows the optimal MDP policy for making adaptation decisions. Using deterministic planning, since hybrid planning makes a quick decision of adding a server of type-C at the time step t_2, it has higher utility than the slow planning approach between the time steps t_4 and t_6. Comparing the hybrid planning and the fast planning approaches, since from t_3 onward the hybrid planning follows an optimal MDP policy, the aggregate utility for the hybrid planning is higher than the fast planning approach.

2) Random Server Crash: To validate the effectiveness of the hybrid planning approach in case of a random hardware failure, we simulated a server crash by removing the type-C server at the point shown in Fig. 4. We chose this particular point because a server crash here results in a constraint violation, which triggers the deterministic planner. The server crash happens at time step t_2 shown in Fig. 7. When high response time is observed at time step t_3, deterministic planning instantly suggests an adaptation tactic of adding a type-C server, which becomes active at the time step t_5. Moreover, at time step t_3, the deterministic planning suggests a tactic for optimal distribution of workload between the other two active servers (i.e., one each of type-A and type-B), which improves the utility between the time steps t_3 and t_4.

However, even though the MDP planning is triggered at time t_3, due to the planning delay, the policy is not ready until time t_4. Therefore, utility remains low compared to the deterministic planning approach for the initial three evaluation cycles of the planning horizon. For this scenario, the performance of the hybrid planning is the same as the deterministic planning since both execute the same set of adaptation tactics for the evaluation cycle. However, the hybrid planning did not perform worse than the deterministic planning approach.

C. Threats To Validity

There are various parameters such as the length of the MDP planning horizon and utility function, which could impact the final results. We have explained the rationale for the length of the planning horizon and utility function, which seems reasonable. Another threat to validity use of a simulated cloud setting rather than a real one. To minimize the impact of using the simulated setup, we used a realistic request arrival trace as an input.

The accuracy of our time-series predictor is another factor that could impact the final results. Mostly, our time-series predictor was able to anticipate future interarrival times with reasonable accuracy. However, if predictions would have often been wrong, the MDP policy based on those predictions would not have been optimal. Moreover, the policy would frequently fail to take over the execution from a deterministic plan. In such situations, the performance of the hybrid planning might be worse compared to using the deterministic planning alone.

The simplifying assumption that MDP planning time is less than an evaluation cycle could be seen as another threat to

validity. However, we do not see it as a limitation to our approach, since MDP planning allows us to extract a sub-optimal MDP policy.[8] To elaborate further, if the MDP planning time is more than τ, a sub-optimal MDP policy could be extracted from the MDP planner at the beginning of the next evaluation cycle and passed as an input (π) to the $hybrid\_planning$ algorithm. These sub-optimal policies could be better than the deterministic plans. However, extracting a sub-optimal policy requires some minimum amount of processing that might not be quick. As future work, we plan to explore the cases where MDP planning time is greater than the evaluation cycle.[9]

In our experimental cloud setting, if the fast planning makes a sub-optimal decision in the short term, an optimal plan produced by the slow planning is able to improve the utility in the long term: there is an opportunity to make up for a sub-optimal rapid decision. However, this might not be true in some domains, where a sub-optimal decision could lead to an irreparable failure state. This leads us to the future challenge of providing formal guarantees about the hybrid planning approach, in particular-about the outcomes of plans generated by a fast planner.

VI. Related Work

A. Automated Planning

The AI community has been working towards finding better algorithms and heuristics to deal with the issue of planning delay. There have been several works striving to improve planning time for deterministic domains [30][2][3][1].

For non-deterministic and probabilistic domains, the state-of-art planning approaches such as MDP [20], POMDP [22], are generally slow in terms of planning time. Various heuristics have been suggested to improve the planning time for MDP and POMDP planning [20][36]. However, long planning time for MDP and POMDP planning is still an open problem.

Researchers also suggested a broad category of planning algorithms, based on the idea of incremental planning, known as *anytime planning*: the planning process can be interrupted at any time to get a sub-optimal plan; however, more planning time leads to better plans [13]. Hybrid planning is close to anytime planning in the sense that hybrid planning uses the execution time of a deterministic plan to find better plans using MDP planning. However, hybrid planning is different because it manipulates the state space to reduce planning time. Moreover, the application of anytime planning depends on the algorithm being used for planning: not all planning algorithms are iterative in nature.

Our hybrid planning approach of combining multiple decision making approaches has been explored by the AI community. Hayes-Roth combined multiple planners to complete different sections of an abstract plan skeleton. The combined planners operate in a hierarchical fashion to solve different sub-problems [39], which is different from the hybrid planning approach.

B. Architectural Frameworks to Support Automated Planning

To support automated planning, various execution frameworks have been proposed [8]. For architecture-based self-adaptation, Kramer and Magee proposed a layered architecture [6] inspired by Gat [7], which aims at dealing with the problem of planning delay through hierarchical decomposition of the planning domains. Tajali et al. extended the hierarchical design by suggesting two types of planning: application planning and adaptation planning [4]. Since hierarchical decomposition of a planning domain helps in reducing the planning state-space, the layered architecture helps to bring down the planning time. However, planning delay also depends on the planning approach deployed at each layer. The hybrid planning approach complements the layered architecture since it could be deployed at each layer to deal with the tradeoff between timeliness and quality of plans.

Quite different from the layered architectures, Musliner et al. proposed a framework that ensures the execution of tasks in a specified deadline [9]. However, unlike hybrid planning, this framework requires hard deadlines to be specified in the planning specification.

C. Automated Planning in Self-Adaptive Systems

Automated planning has been used successfully to determine a sequence operations to go from the initial architectural configuration to the desired architectural configuration [25][37]. In the context of self-adaptive systems, different automated planning approaches have been used to generate repair plans at run time. For instance, Kim et al. [23] and Amoui et al. [24] used reinforcement learning, Moreno et al. [15] used MDP planning, Cámara et al. [27] used a variation of MDP planning known as stochastic multi-player games, Sykes et al. [10] used model-checking for searching the state space, and Zoghi et al. [26] implemented a variation of the hill-climbing algorithm. Generally, the focus of this existing work has been to demonstrate that automated planning could be useful in self-adaptive software systems. Therefore, the existing work either ignores the timing concerns or tends to focus on domains where timing is not a first-class concern.

The hybrid planning approach is different from existing work since we combine two automated planning approaches to deal with the trade-off between quick and optimal planning. The closest approach is used by the FF-Replan planner, which solves a relaxed version of a planning problem by transforming the planning domain with uncertain action outcomes into a deterministic domain by considering only the most probable action outcome [14]. In our case, we transform the probabilistic planning domain into a deterministic planning domain by ignoring uncertainty in the environment. From an experimental perspective, our work is close to Moreno et al. [15]. However, their work focuses on the problem of tactic execution time rather than planning time, which is considered negligible in their experiments.

VII. Conclusion and Future Work

For decision-making in self-adaptive systems, this paper presents a novel hybrid planning approach to deal with potentially conflicting requirements of timeliness and optimality of adaptation plans. Our experiments show that the hybrid planning approach can improve overall utility of self-adaptive systems by finding a right balance between timeliness and optimality.

Although the hybrid planning approach appears to be effective in the cloud computing domain, the approach has certain

[8]MDP planning algorithms such as $value\text{-}iteration$ and $policy\text{-}iteration$ are "Anytime" in nature, since these algorithms monotonically converge (with time) to the optimal MDP policy [13] [20].

[9]This work might require to support the extraction of partial policies in the PRISM tool or develop an MDP planner.

limitations that need to be addressed in the future. Going forward, we plan to validate the approach in safety critical domains (e.g. unmanned aerial vehicles), which require formal guarantees about the adaptation plans generated by a hybrid planner.

The success of hybrid planning depends on the invoking condition of the constituent planning approaches. In our experiments, we considered the response time constraint violation as a trigger for fast planning. However, to apply hybrid planning more broadly, we need to develop a general theory for deciding when to trigger a particular constituent planning approach, again an area for future work.

In our experiments, we combined deterministic planning with MDP planning since the structure of the MDP policy helps in coordinating between the two planning approaches. Going forward, we would like to explore other coordination mechanisms that could be used to combine different planning approaches, such as POMDPs.

ACKNOWLEDGMENTS

This work is supported in part by awards N000141310401 and N000141310171 from the Office of Naval Research (ONR), and FA87501620042 from the Air Force Research Laboratory (AFRL). Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ONR, AFRL, DARPA or the U.S. Government. This material is also 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. Please see Copyright notice for non-US Government use and distribution. (DM-0003199).

REFERENCES

[1] Alfonso Gerevini, Alessandro Saetti, Ivan Serina, "LPG-TD: a Fully Automated Planner for PDDL2.2 Domains" (short paper), in 14th Int. Conference on Automated Planning and Scheduling (ICAPS-04), booklet of the system demo section, Whistler, Canada, 2004.

[2] B. Nebel, The FF Planning System: Fast Plan Generation Through Heuristic Search, in: Journal of Artificial Intelligence Research, Volume 14, 2001, Pages 253 - 302.

[3] Blai Bonet, Hector Geffner, Planning as heuristic search, Artificial Intelligence 129 (2001) 5-33

[4] Hossein Tajalli, Joshua Garcia, George Edwards, and Nenad Medvidovic, PLASMA: A Plan-based Layered Architecture for Software Model-driven Adaptation, ASE-10, September 20-24, 2010, Antwerp, Belgium

[5] Shang-Wen Cheng, David Garlan, Stitch: A language for architecture-based self-adaptation, The Journal of Systems and Software 85 (2012) 2860-2875

[6] Jeff Kramer and Jeff Magee, Self-Managed Systems: an Architectural Challenge, Future of Software Engineering(FOSE'07)

[7] E. Gat, Three-layer Architectures, Artificial Intelligence and Mobile Robots, MIT/AAAI Press, 1997.

[8] David Kortenkamp, Reid Simmons, Handbook of Robotics, Chapter 8: Robotic Systems Architectures and Programming

[9] David Musliner, Ed Durfee and Kang Shin, "World Modeling for Dynamic Construction of Real-Time Control Plans", Artificial Intelligence, 74:1, 1995.

[10] Daniel Sykes, William Heaven, Jeff Magee, Jeff Kramer, Plan-Directed Architectural Change For Autonomous Systems, Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

[11] Carlos Eduardo da Silva, Rogèrio de Lemos, Dynamic Plans for Integration Testing of Self-adaptive Software Systems, SEAMS, 2011, Waikiki, Honolulu, HI, USA

[12] Veloso, M. (1994). Planning and Learning by Analogical Reasoning. Number 886 in Lecture Notes in Computer Science. Springer, Berlin.

[13] Shlomo Zilberstein, Using Anytime Algorithms in Intelligent Systems, American Association for Artificial Intelligence. 0738-4602-1996

[14] Sungwook Yoon, Alan Fern, Robert Givan, FF-Replan: A Baseline for Probabilistic Planning, American Association for Artificial Intelligence, 2007

[15] Gabriel A. Moreno, Javier Cámara, David Garlan, Bradley Schmerl, Proactive Self-Adaptation under Uncertainty: A Probabilistic Model Checking Approach, ESEC/FSE-15, August 30 - September 4, 2015, Bergamo, Italy

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

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

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

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

[20] Mausam, Andrey Kolobov, Planning with Markov Decision Processes: An AI Perspective, Synthesis Lectures On Artificial Intelligence and Machine Learning.

[21] Dana Nau, Malik Ghallab, Paolo Traverso, Automated Planning: Theory and Practice

[22] Leslie Pack Kaelbling, Michael L. Littman, Anthony R. Cassandra, Planning and Acting in Partially Observable Stochastic Domains, Journal of Artificial Intelligence Research, Volume 101, Issues 1-2, May 1998, Pages 99-134

[23] Dongsun Kim and Sooyong Park, Reinforcement Learning-Based Dynamic Adaptation Planning Method for Architecture-based Self-Managed Software, SEAMS'09, Vancouver, Canada

[24] Mehdi Amoui, Mazeiar Salehie, Siavash Mirarab, Ladan Tahvildari, Adaptive Action Selection in Autonomic Software Using Reinforcement Learning, Forth International Conference on Autonomic and Autonomous Systems

[25] Naveed Arshad, Dennis Heimbigner, Alexander. L Wolf, Deployment and Dynamic Reconfiguration Planning For Distributed Software Systems, Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'03)

[26] Parisa Zoghi, Mark Shtern, Marin Litoiu, Hamoun Ghanbari, Designing Adaptive Applications Deployed on Cloud Environments, ACM Transactions on Autonomous and Adaptive Systems, Vol. 10, No. 4, Article 25, Publication date: January 2016.

[27] Javier Cámara, David Garlan, Bradley Schmerl, Ashutosh Pandey, Optimal planning for architecture-based self-adaptation via model checking of stochastic games, SAC 2015: 428-435

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

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

[30] Avrim L. Blum, Merrick L. Furst. Fast Planning Through Planning Graph Analysis. Artificial Intelligence, 90:281-300, 1997

[31] G. Linden, "Make Data Useful," Amazon, 2009.

[32] Daniel S. Weld, An Introduction to LeastCommitment Planning, AI Magazine Volume 15 Number 4 (1994)

[33] Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley R. Schmerl, Peter Steenkiste: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. ICAC 2004: 276-277

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

[35] Mor Harchol-Balter, Performance Modeling and Design of Computer Systems: Queueing Theory in Action

[36] Trey Smith and Reid Simmons, Heuristic Search Value Iteration for POMDPs, CoRR abs/1207.4166

[37] Jeffrey M. Barnes, Ashutosh Pandey, and David Garlan, Automated Planning for Software Architecture Evolution, ASE 2013, Palo Alto, USA

[38] Daniel Kahneman, Thinking, Fast and Slow

[39] Barbara Hayes-Roth and Frederick Hayes-Roth, A cognitive model of planning, Cognitive Science 3, 275-310 (1979)