2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)

Towards a Formal Framework for Hybrid Planning in Self-Adaptation

Ashutosh Pandey, Ivan Ruchkin, Bradley Schmerl, and Javier Cámara
Institute for Software Research
Carnegie Mellon University, Pittsburgh, PA
ashutosp@cs.cmu.edu · iruchkin@cs.cmu.edu · schmerl@cs.cmu.edu · jcmoreno@cs.cmu.edu

***Abstract*—Decision-making approaches in self-adaptation face a fundamental trade-off between quality and timeliness of adaptation plans. Due to this trade-off, designers often have to make an offline compromise between finding adaptation plans quickly and finding closer-to-optimal plans that demand longer computation times. Recent work has proposed that *hybrid planning* can resolve this trade-off dynamically, achieving higher utility than either fast or slow approaches individually. The promise of hybrid planning is to combine multiple decision-making approaches at run time to produce adaptation plans of the high quality within given time constraints. However, the diversity of decision-making approaches makes the problem of hybrid planning complex and multi-faceted. This paper advances the theory of hybrid planning by formalizing the central concepts and four sub-problems of hybrid planning. This formalization can serve as a foundation for creating and evaluating hybrid planners in the future.**

I. Introduction

A typical control loop in many self-adaptive software systems has four computational components: Monitoring-Analysis-Planning-Execution (MAPE) [7]. Prior research has proposed multiple approaches for the planning component to provide decision-making at run time. Frameworks such as Rainbow [9] apply case-based reasoning, solving new problems based on solutions to similar problems from the past. When adaptation is needed, Rainbow chooses an adaptation strategy (i.e., a plan) from a predefined repertoire, which was created at design time by domain experts based on their past troubleshooting experience [3]. In contrast to building a repertoire offline, automated planning techniques (e.g., model-checking [5], reinforcement learning [12], and genetic algorithms [11]) have been explored to generate adaptation plans at run time.

For any decision-making approach[1], quality and timeliness (of adaptation plans) are conflicting requirements. Decision making, in essence, is a search process performed over the space of possible decisions — more complete searches provide better quality guarantees, but require more time to complete. Hence, a planner can either provide a sub-optimal plan at the moment when it is needed, or provide a higher quality plan, risking it being late. For instance, Rainbow provides fast and (potentially) sub-optimal decisions because it gives greater weight to strategies that have worked in the past, and it is difficult to have a predefined strategy for unforeseen scenarios. Alternatively, in a non-emergency situation, a slower deliberative approach may be chosen that fully explores a large decision space to provide an optimal or near-optimal plan. Some self-adaptive systems need to resolve the quality-timeliness conflict at run time: urgent circumstances demand timely actions, but over the long term the performance should be optimized with a deliberative planning. For instance, Amazon Web Services (AWS) are required to maintain an up-time of at least 99.95% in any monthly billing cycle as per the service level agreement, balancing it with other concerns such as cost minimization.[2] The perceived effectiveness of such systems would drop drastically if their service-level constraints were violated. In case of a constraint violation, a rapid response is required to keep the system in a desirable state (for AWS, maintaining availability). However, to maintain a long-term quality, the adaptation plan should be as close to optimal as possible, by considering other metrics as well.

To provide a run-time balance between quality and timeliness, researchers proposed algorithms to improve the performance of a search process [8] and heuristics to reduce a search space [1] [2] [6]. However, these solutions are often non-trivial to develop, yet not generalizable across multiple domains.

In contrast to system-specific solutions, our previous work proposed a general *hybrid planning* approach [13] to balance quality and timeliness at run time. Hybrid planning combines several off-the-shelf decision-making approaches to activate them as necessary, ideally using the most appropriate approach in each situation. The key idea is to use a fast decision-making approach to handle an immediate problem, but simultaneously use a slow approach to provide an optimal solution, merging the plans at run time. This interleaving of approaches reaps the benefits of both worlds: providing plans quickly when the timing is critical, while allowing optimal plans to be generated when the system has sufficient time to do so.

Even though hybrid planning is a promising idea that is potentially applicable to a wide variety of domains, its successful implementation faces substantial challenges, which have not yet been addressed — or even fully explored. It is difficult to identify the conditions of compatibility between planners, how planners need to be configured (e.g., how to choose the planning horizon), and when to stop using one plan and start using another. Moreover, even if some implementation of a hybrid planner overcomes these obstacles, it is not clear how to systematically evaluate such implementations. Hybrid planners are sometimes compared favorably to individual planners, but that is a relatively conservative benchmark. Currently, any

[1]We use the term "decision-making approach" in a broad sense, to refer any approach that could be used to determine adaptation plans, including strategy selection, condition-action pairs, and planning.

[2]aws.amazon.com/ec2/sla

978-1-5386-1550-8/17 $31.00 © 2017 IEEE
DOI 10.1109/SEAMS.2017.14

evaluation is difficult because we lack a fundamental description of the ideal behavior of a hybrid planner.

This paper addresses the complexity of the hybrid planning problem by splitting it into *four subproblems:* (i) selecting scenarios to plan for, (ii) assessing planners on these scenarios, (iii) deciding what plans to combine, and (iv) selecting the optimal sequence of these plans. We give formal definitions of these tasks, thus taking the first step towards a principled theory of hybrid planning. Our *a posteriori* formalization can guide, evaluate, and compare hybrid planner implementations that approximate the ideal solutions to each subproblems.

We start with a motivating example of a cloud-based self-adaptive system. Section II describes the foundational concepts, used in Section III to formalize the four subproblems of hybrid planning. The paper concludes with a discussion of our model's generality and its usage for evaluation of hybrid planners.

A. Motivating Example

To explain the formal framework for hybrid planning, we use a version of a cloud-based self-adaptive system as an example. The system has a three-tiered architecture: a presentation tier, an application tier, and a database tier. The workload on the system depends on the user request arrival rate, which is uncertain since it depends on external demand.

The system's objective is to maximize utility, which depends on the penalty for response time and the cost of active servers. We assume there is a penalty for each request with a response time above some threshold. If response time is higher when averaged across users, adaptation is needed. However, once response time is under control, the system should bring down the operating cost by minimizing the number of active servers.

II. Foundational Concepts

This section defines the basic concepts needed to formalize hybrid planning.

Definition 1 (State)**.** A *state* s is a vector of values of the system's and environment's variables. Time is considered as a state variable. We denote the set of states by S.

Since time is a state variable, S is a potentially infinite set. Moreover, time imposes an implicit total order on states in S.

Definition 2. The function τ returns the time variable of a state. Formally, $\tau : S \rightarrow \mathbb{R}_{>0}$

Definition 3 (Utility of state)**.** The *utility of a state* is defined as $U_s : S \rightarrow \mathbb{R}$, which is a function that maps state s to its valuation.

In this paper, we use the *a posteriori* notion of utility (i.e., assessed after an execution). Our formalization propagates the definition of utility from the ground truth (utility of a particular state in a real system) to abstract notions that the MAPE loop manipulates (e.g., planners). We use this to thus create a formal underpinning for every planning decision of a self-adaptive system, rooted in the utility that this action leads to.

Definition 4 (Execution)**.** An *execution* e is a potentially infinite sequence of states: $e \stackrel{\text{def}}{=} \langle s_1, s_2, \dots \rangle$. We designate a set of executions by E.

We allow infinite executions to model reactive systems that can execute indefinitely. However, to encode goal-oriented systems, infinite sets and sequences can be made finite.

Definition 5 (Partial execution)**.** For an execution e such that $e \stackrel{\text{def}}{=} \langle s_1, \dots, s_j, \dots, s_n \rangle$, a *partial execution* e_p is a prefix of e ending with s_j where $1 \leq j \leq n$. That is, $e_p \stackrel{\text{def}}{=} \langle s_1, \dots, s_j \rangle$.

Definition 6 (Utility of execution)**.** The *utility of an execution* is defined as $U_e : E \rightarrow \mathbb{R}$, which is a function that maps execution e to its valuation.

Even though an execution is a potentially infinite sequence of states, we assume its utility would be a finite value. As an example, suppose U_e is defined as the utility of a state (in an execution) with the maximum utility (among the states in the execution); here, the utility of an execution would be a finite value. In our model, we abstract away the particular function representing the utility of executions.

We model hybrid planning in the *a posteriori* fashion: some of our utility functions require perfect knowledge of the future to be computed. Although an obstacle for direct implementations, this model is beneficial for formalizing the problem and its idealized solution. In fact, by using information about the future (e.g., how much utility is accrued from an execution), we can establish a theoretical baseline for evaluation of downstream engineering solutions. These solutions will use relaxations (e.g., a priori utility or expected utility) of our utility notion to construct approximations of the idealized solution.

Definition 7 (Transition, action, and event)**.** *State transitions* are characterized by a transition function $T : S \times A \times Z \rightarrow S$, where A is a set of the *system's actions*, and Z is a set of *external events*. An element $\perp$ represents an empty action/event and is present in both sets: $A \cap Z = \{\perp\}$.

A self-adaptive system is characterized by controllable actions (e.g., adding/removing a server) and uncontrollable events (e.g., an arrival of a user request). Both actions and events cause state transitions. T captures both asynchronous (some action along with $\perp$ event, or vice versa) and synchronous (neither the action nor the event are $\perp$) interactions of the system and its environment. Since the model is *a posteriori*, the outcomes of actions and events are deterministic post transition. Thus, T is a function instead of relation.

In our model, we only consider *Markovian* domains, where the state after a transition only depends on the current state — not on the sequence of states that preceded it [18]. We also assume that all transitions take time: the future state's time is always larger than that of the previous state's.

Definition 8 (Plan)**.** A *plan* π is a total function $\pi : S \rightarrow A$. A mapping from a state $s \in S$ to an action $a \in A$ suggests a to be executed in s. We denote a set of plans as Π.

Definition 8 is general enough to capture different types of plan. For instance, universal plans such as MDP policies could be linked directly to the state-action mapping provided by the function π [8]. This definition also captures sequential plans since the time state variable can be used to maintain the ordering of actions during execution.

Definition 9 (Environment)**.** The *environment* is a function $o : S \to Z$ encoding which event happens in each state. A set of environments is designated as O.

Our abstraction of o for events is analogous to π for actions.

Definition 10 (Realization)**.** The *realization* function $\mathcal{R} : \Pi \times O \times S \to E$ maps a plan π, an environment o, and an initial state $s^i \in S$ to the execution e produced by the system executing in those conditions. That is, $\mathcal{R}(\pi, o, s^i) = e$.

$\mathcal{R}$ requires o and π as an input since both influence transitions.

Definition 11 (Utility of plan)**.** The *utility of a plan* π given an environment o and an initial state s^i is a function $U_\pi : \Pi \times O \times S \to \mathbb{R}$ that returns the utility of that plan's realization. That is, $U_\pi(\pi, o, s^i) \stackrel{\text{def}}{=} U_e(\mathcal{R}(\pi, e, s^i))$.

By linking the utilities of plans and executions, we have extended the ground truth to the internal reasoning of planners. This bridge lets us establish utility-based comparison of concepts that normally exist before execution happens. Thus, we trade implementability for a theoretical way of putting value on planning decisions.

Definition 12 (Planning problem)**.** A *planning problem* ξ is a tuple (S, s^i, A, T, o, U_e), where $s^i \in S$ is the initial state. Solving a planning problem means optimizing U_e by providing a plan for given S, s^i, A, T, and o. A set of planning problems is denoted by Ξ.

Self-adaptive systems have flexibility in choosing adaptation scenarios to investigate. For instance, the system can choose its lookahead horizon: should it consider a future of one minute or one hour ahead of the current moment [17]? The space of such decisions is encoded as Ξ.

Definition 13 (Planner)**.** A *planner* is a function $\rho : \Xi \to \Pi$ that solves a planning problem ξ and produces a plan π. We designate a set of planners by Ψ.

Most planner implementations allow numerous customizations. We formalize these customizations as individual planners in Ψ^ξ without loss of generality: each planner is always evaluated independently and with respect to some ξ.

Definition 14 (Problem-Planner Compatibility Relation)**.** A problem ξ and a planner ρ are *compatible* if ρ can solve ξ, denoted $(\xi, \rho) \in \Upsilon$, where $\Upsilon : \Xi \leftrightarrow \Psi$ is a problem-planner *compatibility relation*. Given ξ, $\Psi^\xi \subseteq \Psi$ is a set of planners that are compatible with ξ (i.e., $\Upsilon[\xi] \stackrel{\text{def}}{=} \Psi^\xi$).

In practice, some planners (e.g., deterministic ones) are not applicable to problems (e.g., ones with non-deterministic transitions) that do not match their input format or algorithmic parameters. Nevertheless, often several planners can solve the same problem. For instance, several decision-making approaches are applicable in self-adaptive cloud systems: case-based reasoning [9], automated planning [14], and reinforcement learning [15]. Υ encodes such restrictions, naturally constraining the domain of planner functions in Definition 13.

Definition 15 (Utility of partial execution)**.** The *utility of partial execution* U_{e_p} for a pair of a problem and a planner is a function $U_{e_p} : \Psi^\xi \times \Xi \times S \to \mathbb{R}$ maps a planner ρ, a planning problem ξ, and a state s_{end} to the utility of the partial execution e_p induced by them: $U_{e_p}(\rho, \xi, s_{end}) \stackrel{\text{def}}{=} U_e(e_p)$.

Definition 16 (Utility of planner)**.** The *utility of a planner* ρ, given a compatible ξ, is a function $U_\rho : \Psi \times \Xi \to \mathbb{R}$ that returns a real number — the performance of plan $\rho(\xi)$. This function is defined via the utility function for plans: $U_\rho(\rho, \xi) \stackrel{\text{def}}{=} U_\pi(\rho(\xi))$.

Definition 17 (Utility of planning problem)**.** The *utility of a planning problem* is a function $U_\xi : \Xi \times \Psi \to \mathbb{R}$ that, given ξ and Ψ^ξ, returns the maximum utility among all Ψ^ξ:

$$U_\xi(\xi, \Psi^\xi) \stackrel{\text{def}}{=} \max_{\rho \in \Psi^\xi} U_\rho(\rho, \xi)$$

Let us illustrate these definitions with the cloud-based system from Section I-A. To adapt that system, Ψ contains two planners, based on Markov Decision Processes (MDP, ρ_{mdp}) [18] and case-based reasoning (CBR, ρ_{cbr}). MDP is slow but optimal, suited for ξ with a predictable and stable o. CBR is fast but sub-optimal, suited for ξ with an unpredictable and rapidly changing o. To carry out hybrid planning, the self-adaptive system will find the best combinations of these two planners by selecting appropriate ξ in Ξ and assigning them to the most fitting planner in advance (to account for their planning delays).

III. Decomposition of the Hybrid Planning Problem

We start with a central concept of the paper — a hybrid plan.

Definition 18. [Hybrid plan] A *hybrid plan* is a function $\omega : S \to A$ based on partitioning of the full state space S into n partitions S_i, each governed by a planner ρ_i.

$\exists n : \mathbb{N} \cdot \forall i : 1..n \cdot$ — There exists a number
$\exists \pi_i : \Pi, S_i \subseteq S \cdot S_i \neq \varnothing \ \wedge$ — of plans and state partitions
$(\bigcup_{j:1..n} S_j = S) \wedge (\bigcap_{j:1..n} S_j = \varnothing)$ — that partition the state space such that three conditions hold:

Condition 1: actions in partitions are governed by their plans.

$$\forall s : S_i \cdot \omega(s) = \pi_i(s).$$

Condition 2: plans come from solving planning problems.

$$\exists \xi_i : \Xi,\ \rho_i : \Psi \cdot\ \pi_i = \rho_i(\xi_i).$$

Condition 3: partitions are totally ordered in time.

$$\forall k, l : 1..n, s_1 : S_k, s_2 : S_l \cdot$$
$$k < l \implies \tau(s_1) < \tau(s_2),$$

For example, a hybrid plan that switches between MDP and CBR planners in response to an emergency would have two partitions: S_{mdp} containing the times before the emergency and S_{cbr} containing the times after. A plan from ρ_{mdp} would provide actions for states in S_{mdp}, and a plan from ρ_{cbr} would provide actions in S_{cbr}.

The rest of this section describes the theoretical steps to obtain ω through S_i, π_i for $i : 1..n$ in Definition 18.

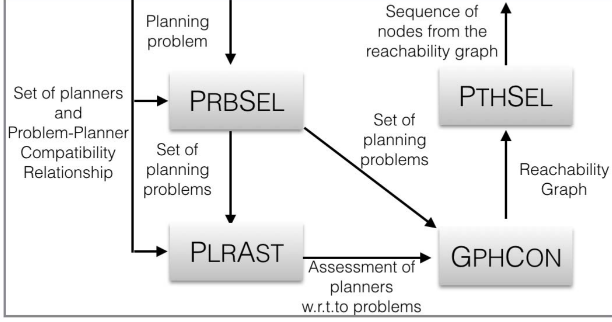

Fig. 1. Decomposition of the hybrid planning problem.

Definition 19 (Hybrid Planning Problem)**.** The *Hybrid Planning Problem* (HPP) is, given an initial planning problem ξ^i, a set of planners Ψ^ξ, and a compatibility relation Υ, find a hybrid plan that maximizes the utility of execution U_e.

This paper's central contribution is a decomposition of HPP into four sub-problems (starting from the end, see Figure 1):

1) *Path Selection* (PTHSEL): what is the sequence of planner invocations on planning problems that yields the maximum utility?
2) *Reachability Graph Construction* (GPHCON): what planning problems are reachable by solving other problems?
3) *Planner Assessment* (PLRAST): what are the quality and timeliness of each planner on a given planning problem?
4) *Problem Selection* (PRBSEL): what planning problems to solve?

A. Path Selection

The *Path Selection* (PTHSEL) subproblem is, informally, to find what sequence of plans from different planners yields the highest utility. This sequence of plans constitutes a hybrid plan ω according to Definition 18. The total number of plans in the sequence would represent the value of *n* (possibly infinite). A plan π_i (ith in the sequence) can, given an environment from the planning problem, be realized to an execution, which in turn can be mapped to a sequence of states. Therefore, each plan π_i can be mapped to the sequence of states.

The question posed in PTHSEL is where one execution ends in the sequence of states, and another one begins. To answer this question, one has to provide a sequence of partitions S_i that determine each plan's/execution's boundary, according to Condition 1 of Definition 18. To satisfy Condition 2, we map these partitions to planning problems and planners. To construct this mapping, we formalize an input structure to PTHSEL that encodes potential choices of problems and planners.

Definition 20 (Reachability graph)**.** A *reachability graph* Γ is a directed graph defined as a tuple $(V, \mathcal{E}, V^i)$. V is a set of nodes, where each node v is a tuple (ξ, ρ, d) combining a problem, a planner, and a deadline d, which is the worst-case time instant when ρ needs to be invoked on ξ (detailed in Section III-C). The set of edges ($\mathcal{E} \subseteq V \times V$) describes reachability between nodes in terms of executions: an edge $\epsilon = (v_1, v_2)$ means that executing $v_1.\rho(v_1.\xi)$ with $v_1.\xi.o$ reaches s^i of $v_2.\xi$. Initial nodes ($V^i \subseteq V$) indicate the potential starts of executions in Γ.

Paths (i.e., sequences of edges) in Γ mimic executions of the system, guided by a sequence of plans. A path indicates a sequence of switches between planning problems, which can be mapped to plans π_i and partitions S_i in Definition 18. PTHSEL selects a path based on the utility of its execution. We introduce several auxiliary concepts to express that selection.

Definition 21 (Edge execution)**.** *Edge execution* is a function $\eta : \mathcal{E} \to E$ that maps an edge ϵ to its execution e. Edge $\epsilon = (v_1, v_2)$ maps to its execution from the initial state of planning problem in the first node to the initial state of the planning problem in the second node: $\eta(\epsilon) = \mathcal{R}(v_1.\rho(v_1.\xi),\ v_1.\xi.o,\ v_1.\xi.s^i)$ which is truncated at $v_2.\xi.s^i$.

Since $\mathcal{R}$ is truncated at $v_2.\xi.s^i$, an edge execution could also be represented as a partial execution $e_p = \langle v_1.\xi.s^i, \ldots, v_2.\xi.s^i \rangle$.

Definition 22 (Path execution)**.** *Path execution* is a function $\eta : \mathcal{E}^n \to E$ maps a path to its execution. A path $\kappa = \langle \epsilon_1, \ldots, \epsilon_n \rangle$ maps to an execution composed of concatenation of edges' executions: $\eta(\kappa) = \eta(\epsilon_1) \frown \ldots \frown \eta(\epsilon_n)$.

The utility of a path in Γ builds upon the utilities of its edges, which in turn build on the utilities of its executions.

Definition 23 (Utility of edges and paths)**.** The *utility of an edge* ϵ is a function $U_\epsilon : \mathcal{E} \to \mathbb{R}$ that maps ϵ to the utility of the edge's execution. Formally, $U_\epsilon(\epsilon) \stackrel{\text{def}}{=} U_e(\eta(\epsilon))$. Similarly, the *utility of a path* $\kappa = \langle \epsilon_1, \ldots, \epsilon_n \rangle$ is a function $U_\kappa : \mathcal{E}^n \to \mathbb{R}$ that is defined as $U_\kappa(\kappa) \stackrel{\text{def}}{=} U_e(\eta(\kappa))$.

Since execution for an edge $\epsilon = (v_1, v_2)$ could be represented as a partial execution, $U_e(\eta(\epsilon)) \stackrel{\text{def}}{=} U_{e_p}(v_1.\xi,\ v_1.\rho,\ v_2.\xi.s^i)$. We are now ready to formalize PTHSEL.

Definition 24 (PTHSEL)**.** The *Path Selection* (PTHSEL) subproblem is, given a reachability graph Γ, to find a maximal-utility path starting from an initial node:

$$\text{PTHSEL}(\Gamma) \stackrel{\text{def}}{=} \arg\max_{\kappa \in \mathcal{E}^n} U_\kappa(\kappa).$$

Among the four sub-problems of HPP, PTHSEL is the last one to be solved before a hybrid plan is ready. Once an optimal path is found, it translates into plans and partitions (equal to the number of nodes in the path) to define a hybrid plan. Due to the strict time ordering of partitions (Condition 3 in Definition 18), past plans cannot be directly reused.

To exemplify PTHSEL, consider a rapid spike in the request arrival rate for the cloud-based system (Section I-A). PTHSEL compares two paths: use ρ_{mdp} indefinitely, or initially use ρ_{mdp} and later switch to ρ_{cbr}. If ρ_{cbr} yields larger utility on average after the spike, PTHSEL would switch to it, thus improving the system's utility compared to the first path.

B. Graph Construction

The purpose of the *Graph Construction* (GPHCON) subproblem is to build reachability graph Γ to be used by PTHSEL. As Figure 1 shows, to build this graph GPHCON relies on two inputs: planning problems Ξ that need to be solved (from PRBSEL) and compatible planners Ψ^ξ. For each pair of $\xi \in \Xi$ and $\rho \in \Psi^\xi$, PLRAST provides utility of partial execution U_{e_p} and invocation deadline d.

Nodes of Γ are constructed as follows: For a given ξ, we create a node for each planner ρ compatible with ξ. In each node, we add deadline d of the planner. We repeat this process for each planning problem received from PRBSEL. Thus, we obtain a set of nodes such that each node is a triple (ξ, ρ, d).

An edge could be constructed between a pair of nodes v_1 and v_2, if and only if two conditions are met:

1) *Preemption:* after executing the plan from v_1, the system should reach the initial state of the planning problem in v_2. Only then the plan for v_2 can take over from the previous plan. Formally, $v_2.\xi.s^i = last(v_1.\rho(v_1.\xi))$ where $last$ is a function that returns the end state of a plan execution.
2) *Timing:* the plan in v_2 should be ready once the execution comes to it. Hence, $v_2.\rho$ has to be triggered before the system faces $v_2.\xi$. This needs to be at least the worst case planning time for $v_2.\rho$ to solve $v_2.\xi$. Mathematically, the only reason an early enough time will not be found is when $t < 0$. Therefore, the condition for ρ having enough time before its execution is: $d(\rho) > \sum_{\epsilon_i \in \langle \epsilon_0, \ldots \epsilon_n \rangle} duration(\mathcal{R}(edge_i))$, where function $duration$ returns the duration of execution for an edge.

Now, we are ready to define GPHCON formally.

Definition 25 (GPHCON)**.** The *Graph Construction* (GPHCON) problem is, given planning problems Ξ, planners Ψ^ξ, respective utility U_{e_p}, and deadline d functions, find a reachability graph Γ with edges satisfying the preemption and timing conditions.

To build a reachability graph, PRBSEL provides Ξ and Ψ^ξ whereas PLRAST provides U_{e_p} and d. In practice, GPHCON is unlikely to be fully constructed for even moderately sized problems. Therefore, the goal of implementations is to build the most effective subgraph of Γ. The cloud-based self-adaptive system can, for example, place nodes at times of large expected changes in the incoming traffic. Edges can be made probabilistic (based on historic information and heuristics) to avoid requiring exhaustive traversal of the state space.

C. Planner Assessment

The *Planner Assessment* (PLRAST) subproblem is, given planning problem ξ and a set of compatible planners Ψ^ξ, rate the performance of these planners on that problem. These ratings are an essential part of GPHCON (Section III-B), and obtaining them is a difficult and separate subproblem of HPP.

For hybrid planning, we are interested in two aspects of planners' performance: quality and timeliness. We model quality with propagating utility functions (i.e., linked to executions) defined in Section II. For timeliness, we adopt the worst-case model of time: we assume the knowledge of the maximum time needed by a planner for a planning problem. Thus, PLRAST requires finding the worst-case planning time for each planner.

The plan's quality and timeliness are determined by the planner and the planning problem. Therefore, the inputs to PLRAST are the planning problem ξ and a set of compatible planners Ψ^ξ.

For an input planning problem, PLRAST has two outputs: (1) utility function U_{e_p} for partial executions of plans from each compatible planner, (2) deadline $d : \Psi^\xi \times \Xi \to \mathbb{R}_{>0}$, a function that returns the worst-case planning delay for the planning problem with respect to each compatible planner.

Definition 26 (PLRAST)**.** The *Planner Assessment* (PLRAST) *problem* is, given planning problem ξ and compatible planners Ψ^ξ, find the utility of partial execution U_{e_p} and deadline d (ρ) for each planner $\rho \in \Psi^\xi$ on ξ.

To solve PLRAST in practice, one needs to create algorithms to measure utilities and deadlines for planners. To the authors' knowledge, the majority of existing planner implementations do not provide up-front guarantees on either of these two characteristics. Therefore, two general approaches are possible: (i) design new planners with guarantees of quality and timeliness on given planning problems, and (ii) determine the characteristics of existing planners. While (i) is self-evident, (ii) can be accomplished in a number of ways—from theoretical modeling to empirical profiling. This formalization of PLRAST explains how to evaluate such solutions in a uniform way.

D. Problem Selection

The *Problem Selection* (PRBSEL) subproblem is to choose which planning problems should be solved at a given time. At every moment, an infinite number of planning problems can be formulated: according to Definition 12, one can arbitrarily select the initial state, the subset of actions, the subset of the state space, the environment's choices of events, and the utility function. Hence, PRBSEL reduces all possible planning problems to a smaller set of problems that is fed into GPHCON.

As time passes, some initial states become unreachable, and those problems become obsolete. Thus, the set of relevant planning problems changes. Due to delays in planning, proactive planning is needed for problems in the future, so that plans are ready by the time the execution approaches those problems.

Definition 27 (Time-bound Ξ)**.** Given a time t, a *time-bound planning problem set* Ξ_t is a set of planning problems whose initial states have time t: $\Xi_t = \{\xi : \Xi \mid \tau(\xi.s^i) = t\}$.

Each time-bound set Ξ_t needs to be filtered through the compatibility relation Υ: only problems that have at least one compatible planner need to be allowed. The result is a *filtered time-bound set of planning problems*: $\Xi_{t,\Upsilon} = \{\xi : \Xi \mid \Psi^\xi \neq \varnothing\}$.

An optimal hybrid planner executes the plan from the highest-utility planning problem at each moment. For a moment t, we select the problems from $\Xi_{t,\Upsilon}$ that yield the maximum utility. To define an optimal hybrid planner, this subset needs to be found for each possible t. Otherwise, we would have to consider nodes in Γ that are guaranteed to not yield an optimal ω.

The inputs to PRBSEL are the initial problem ξ^i, the set of planners Ψ, and the compatibility relation Υ. The output of PRBSEL is a set of planning problems $\Xi^*_{t,\Upsilon} \subseteq \Xi_{t,\Upsilon}$ that will be solved, and their plans will inform the behavior of the self-adaptive system.

Definition 28 (PRBSEL)**.** The *Problem Selection* (PRBSEL) subproblem is, given the initial planning problem ξ^i, the

set of planners Ψ, and the compatibility relation Υ, select the maximum-utility planning problems with at least one compatible planner for each time instant.

$$\forall t \cdot \Xi^*_{t,\Upsilon} = \arg\max_{\xi \in \Xi_{t,\Upsilon}} U_\xi(\xi)$$

Two obstacles make practical implementation of PRBSEL difficult. First, it is impossible to decide the best planning problem for each time moment. Therefore, implementations of PRBSEL would need a mechanism to choose time moments, for example periodically. The second practical issue for PRBSEL is the mutual dependency between PRBSEL and PLRAST: a problem cannot be evaluated without planners, and planners — without a problem. We accept this dependency in theory, hoping it will be resolved in practice using approximations. For example, one can use machine learning to predict which planners yield the most utility on each problems.

IV. DISCUSSION

We now discuss two characteristics of our hybrid planning formalization: its use for an experimental validation of hybrid planners, and its generality.

A. Evaluation of Hybrid Planners

The proposed formalization uses two idealized notions: *utility* of plans/planners/problems and *reachability* between nodes containing problems and planners. We use the former to measure "goodness" of decisions in subproblems. We use the latter to combine plans into a hybrid plan with guaranteed preemption and timing. While these idealizations are not directly implementable, they do provide a uniform way to evaluate future solutions to the subproblems of hybrid planning.

To estimate the difference between an optimal planner and an implementation, our utility and reachability notions enable an *evaluation workflow*: (1) implement a hybrid planner and a simulation of a system; (2) obtain a hybrid plan ω and execute it with different o, logging complete execution traces; (3) calculate utility of traces according to Definition 6; (4) reconstruct a reachability graph for each scenario; (5) perform what-if simulations to find (a) more optimal or timely paths, (b) other planning problems from $\Xi^*_{t,\Upsilon}$, (c) missing or inaccurate ϵ, and (d) other opportunities for improvement of ω; (6) The identified improvements characterize the delta between the empirical and theoretical utilities.

This is a repeatable evaluation procedure for hybrid planners, grounded in theoretical concepts. It is applicable to a wide variety of planner combinations, including prior work on combining contingency plans [16]. Although such experiments can be computationally expensive, they yield valuable insights into the behavior and potential improvements of hybrid planners.

B. Generality

We acknowledge that while no other formalization of hybrid planning exists, such formalizations might be possible. A distinctive feature of our formalization is its *parsimony*: we use only the essential concepts broadly applicable to planners, and we introduce the least restrictive assumptions that enable precise and substantial subproblems. Below we summarize our assumptions to assess their scope of validity.

Fixed set of planners: we consider realistic situations where planning tools are known before they are executed in a system. This assumption is valid in practically all contexts where planners are used today.

No instantaneous actions: in practice, no action takes exactly zero time to execute. Therefore, this assumption supports the generality of the formal model.

Markovian domain: many of the domains explored by the self-adaptive community are assumed to be Markovian [5], [12], [15] (even though not always explicitly stated). Therefore, even with this assumption, the proposed formal model is applicable to various domains, particularly those investigated by the self-adaptive community.

Instantaneous solutions to subproblems: we consider the delays of actions and planning itself, but not the delays of solving PTHSEL, GPHCON, PLRAST, and PRBSEL. This assumption holds if solving these problems takes negligible time compared to the time scale of planning and execution — or if the solutions are pre-computed offline.

Known worst-case planning time: currently most planners cannot provide a hard guarantee on their planning time. We hope, however, that extensive up-front profiling of planners can lead to strong empirical guarantees on worst-case planning times. The goal of relaxing this assumption opens a promising direction of future work—predictable planners in self-adaptation.

Known utility of states/executions: this assumption holds in most contexts of self-adaptation in software systems, except when experimental data is incomplete or inaccessible. One example is complex cyber-physical systems where the physical state may be difficult to monitor and log entirely.

To summarize, this paper decomposes the sophisticated problem of hybrid planning into four computational subproblems. We expect that, due to the complexity of the problem, hybrid planning implementations will vary in formalisms and algorithms that address the subproblems. The formal definitions offered in this paper can serve as a unifying evaluation framework for practical solutions to hybrid planning.

We would like to encourage the self-adaptive systems research community to actively participate in creating engineering solutions to the subproblems of hybrid planning. The prior work [13], [16] shows that hybrid planning is a promising way to improve self-adaptation, thus increasing the potential for industrial adoption. However, the complexity of hybrid planning creates a possibility for many diverse approaches. Therefore, extensive research is needed to provide efficient, usable, and general approaches to combine multiple planners for self-adaptation.

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.

REFERENCES

[1] J. Hoffmann and B. Nebell, "The FF Planning System: Fast Plan Generation Through Heuristic Search", Journal Artificial Intelligence Research, vol. 14, 2001, pp. 253 - 302.

[2] Blai Bonet and Hector Geffner, "Planning as heuristic search", Artificial Intelligence, vol. 129, 2001, pp. 5-33.

[3] Shang-Wen Cheng and David Garlan, "Stitch: A language for architecture-based self-adaptation", The Journal of Systems and Software, vol. 85, issue 12, December 2012, pp. 2860-2875.

[4] Jeff Kramer and Jeff Magee, "Self-Managed Systems: an Architectural Challenge", in Future of Software Engineering(FOSE'07), 2007©IEEE. doi:[0-7695-2829-5/07]

[5] Daniel Sykes et al., "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, 2007©ACM, ISBN 978-1-59593-721-6/07/0009.

[6] Sungwook Yoon et al., "FF-Replan: A Baseline for Probabilistic Planning", Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007.

[7] J. O. Kephart and D. M. Chess, "The vision of autonomic computing", Computer, vol 36, issue 1, Jan 2003. doi:[10.1109/MC.2003.1160055]

[8] Malik Ghallab et al., "Automated Planning: Theory and Practice", Morgan Kaufmann, May 2004.

[9] Shang-Wen Cheng et al., "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure", 1st International Conference on Autonomic Computing (ICAC 2004), pp. 276-277. doi:[10.1109/MC.2004.175]

[10] Mor Harchol-Balter, "Performance Modeling and Design of Computer Systems: Queueing Theory in Action", Cambridge University Press, February 2003.

[11] Zack Coker et al., "SASS: Self-Adaptation Using Stochastic Search", Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems Florence, Italy 2015, pp. 168-174.

[12] Dongsun Kim and Sooyong Park, "Reinforcement Learning-Based Dynamic Adaptation Planning Method for Architecture-based Self-Managed Software", Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Vancouver, Canada, 2009, pp. 76-85.

[13] A. Pandey et al., "Hybrid planning for decision making in self-adaptive systems", in International Conference on Self-Adaptive and Self-Organizing Systems, ser. SASO 2016, 2016, pp. 12-16.

[14] Javier Cam̀ara et al., "Optimal planning for architecture-based self-adaptation via model checking of stochastic games", 30th ACM/SIGAPP Symposium On Applied Computing (SAC 2015), April 13 - 17, 2015, pp. 428-435.

[15] Barry Porter and Roberto Rodrigues Filho, "Losing Control: The Case for Emergent Software Systems using Autonomous Assembly, Perception and Learning", in International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2016.

[16] M. Beetz and D. McDermott, "Improving robot plans during their execution," in International Conference on AI Planning and Scheduling, AIPS '94, 1994.

[17] Roykrong Sukkerd et al., "Multiscale time abstractions for long-range planning under uncertainty", Proceedings of the 2nd International Workshop on Software Engineering for Smart Cyber-Physical Systems, SEsCPS@ICSE 2016, Austin, Texas, USA, May 14-22, 2016. ACM 2016, ISBN 978-1-4503-4171-4. doi:[10.1145/1235

[18] Mausam and Andrey Kolobov, "Planning with Markov Decision Processes: An AI Perspective", Synthesis Lectures On Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers, June 2012, ISBN:9781608458875.