Submitted for publication.

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*—Approaches to self-adaptation face a fundamental trade-off between quality and timeliness in decision-making. Due to this trade-off, designers of self-adaptive systems often have to find a fixed and suboptimal compromise between these two requirements. Recent work has proposed the hybrid planning approach that can resolve this trade-off dynamically and potentially in an optimal way. The promise of hybrid planning is to combine multiple planners at run time to produce adaptation plans of the highest quality within given time constraints. However, since decision-making approaches are complex and diverse, the problem of combining them is even more difficult, and no frameworks for hybrid planning. This paper makes an important step in simplifying the problem of hybrid planning by formalizing it and decomposing it into four simpler subproblems. These formalizations will serve as a foundation for creating and evaluating engineering solutions to the hybrid planning problem.**

I. Introduction

A typical control loop in many self-adaptive software systems has four computational components: Monitoring-Analysis-Planning-Execution (MAPE), coordinated through knowledge [7]. Prior research has proposed multiple approaches for the planning component to provide decision-making at run time.[1] Frameworks such as Rainbow [12] apply case-based reasoning, solving new problems based on solutions to similar problems seen in the past. When adaptation is needed, Rainbow chooses an adaptation strategy (i.e., a plan) from a repertoire of predefined adaptation strategies 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 [15], and genetic algorithms [14]) have been explored to generate adaptation plans at run time.

For any decision-making approach, quality and timeliness (of adaptation decisions) 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 about the decisions, but require more time to be completed. Hence, a planner can either provide a partially ready plan at the moment when it is needed, risking it being sub-optimal, or provide a fully ready plan, risking it being late. For instance, in the Rainbow framework, fast decisions can be made using case-based reasoning, however decisions could be sub-optimal since it is difficult to have a predefined strategy for unforeseen scenarios. Alternatively, in less urgent situation, a slower approach may be chosen that fully, and more deliberatively, explores a large decision space and provide an optimal or near-optimal plan.

Some self-adaptive systems need to resolve the quality-timeliness conflict at run time: they need timely actions under urgent circumstances, but eventually require deliberative planning to improve their performance over the long term. For instance, Amazon Web Services (AWS) [2] are required to maintain an up-time of at least 99.95% in any monthly billing cycle as per the service level agreement, even though there might be other quality concerns such as cost minimization. The perceived quality for such systems would drop drastically if such service-level constraints were violated. For systems like AWS, in case of a failure a rapid response is required to keep the system in a desirable state (for AWS, maintaining availability). However, to maintain quality of the system, the adaptation plan should be as close to optimal as possible, by considering other metrics such as operating cost.

To provide a run-time balance between quality and timeliness, researchers have proposed algorithms to improve the search [10] and heuristics for reducing the search space [6] [2] [1]. However, these solutions often do not generalize to many scenarios, remain specific to systems for which they were devised.

In contrast to inventing system-specific solutions, previous work proposed a general *hybrid planning* approach [16] to balance quality and timeliness at run time. Hybrid planning combines several off-the-shelf decision-making approaches to activate these approaches 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 on the fly. This interleaving of approaches helped in achieving 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 and 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 planning approaches, 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 overcomes these obstacles, it is not clear how to systematically evaluate that implementation. A particular hybrid planning approach may

[1]The term "decision-making approaches" is used in a broad sense, representing any approach that could be used in planning to determine adaptation plans.

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

perform better than any particular planner, but that is a relatively low standard given limitations of individual planners. Currently, any evaluation is difficult because a fundamental description of the ideal behavior for hybrid planning is lacking.

This paper addresses the complexity and vagueness of the hybrid planning problem by splitting it into *four subproblems:* (i) selecting which scenarios to plan for, (ii) rating available planners on these scenarios, (iii) deciding what subplans can be combined, and (iv) selecting the most optimal sequence of subplans. We give formal definitions to these concepts, thus taking a first step in building a principled theory of hybrid planning. These formal definitions can be used as design guidance and an evaluation method for making hybrid planners. Before developing solutions for hybrid planning, it is necessary to understand the subproblems. Furthermore, our notion of utility can be used as a baseline for evaluating implementations of hybrid planning.

We start with a motivating example of a cloud-based self-adaptive system in the next section. Then we describe the basic concepts of hybrid planning, which are used in Section IV to formalize the four subproblems of hybrid planning. We conclude the paper with a discussion of the model's generality and its usage for evaluation of hybrid planners.

II. Motivating Example

To explain elements of the formal framework defined later, the paper uses a simplistic version of a typical cloud-based self-adaptive system as an example. The system has a typical three layered architecture: a presentation layer, an application layer, and a database layer. The workload on the system depends on the request arrival rate, which is uncertain as it depends on the external demand.

The goal of the system is to maximize the utility, which depends on the penalty for response time, and the cost of active servers. We assume there is a penalty, say P, for each request having a response time above the threshold. Therefore, in case of a high average response time, the system needs to react. However, once response time is under control, the system should execute adaptation tactics to bring down the operating cost i.e., needs to minimize the number of active servers.

III. Foundational Concepts

This section defines basic concepts needed to formalize hybrid planning. We interleave definitions of abstractions (e.g., plans) with utility functions that evaluate these abstractions.

Definition 1 (State). A state s is a vector of values of the system's and environment's variables. We designate the set of states by S.

Definition 2 (Utility of states). The utility of a state is defined as $U : S \to \mathbb{R}$, 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 of 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 hope to thus create a formal underpinning for every planning decision of a self-adaptive system, rooted in utility that this action leads to.

Some of our utility abstractions may be not implementable directly, since they require perfect knowledge of the future. Although an obstacle for practical systems, this circumstance is beneficial for formalizing the problem of hybrid planning and its idealized solution. In particular, by using information about the future (e.g., how much utility was accrued from each state), we can set a firm theoretical baseline for evaluation of downstream engineering solutions. These solutions will use relaxations of our utility notion (e.g., a priori utility or average expected utility) to construct approximations of the idealized solution.

Definition 3 (Execution). An execution e is a potentially infinite sequence of states: $e \overset{\text{def}}{=} \langle s_1, s_2, \dots \rangle$. We designate set of executions by E.

In this paper we allow infinite executions and aggregates of infinities. For implementation purposes, infinite sets and sequences can be made finite.

Definition 4 (Utility of executions). The utility of an execution is defined as $U : E \to \mathbb{R}$, which is a function that maps execution e to its valuation.

Utility of system execution (as well as other utility functions we introduce below) directly builds upon the utility of states in the execution. We abstract away the particular form of this function (e.g., summation or averaging).

Definition 5 (Actions and transitions). An action a is a transition between states. We designate a set of all actions by A. Actions are characterized by the state transition function $T : S \times A \to S$.

Definition 6 (Plan). A plan π is, given a planning problem, a set of state-action tuples (s, a), suggesting a to be executed in s, where $s \in S$ and $a \in A$. Π is a set of all plans. We link a plan π to the execution it produces e through a special linking function $\mathcal{L} : \Pi \to E$; $\mathcal{L}(\pi) = e$.

Definition 7 (Utility of plans). The utility of a plan is defined as $U : \Pi \to \mathbb{R}$, which is a function that maps a plan to its valuation by executing the plan in the system. In other words, $U(\pi) \overset{\text{def}}{=} U(e)$ if $e = \mathcal{L}(\pi)$.

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

Definition 8 (Planning problem). A planning problem ξ is a tuple (S, s^i, A, T, U), where $s^i \in S$ is an initial state.U is a utility function for executions. It will propagate it to plans as per Definition 7. By choosing actions under the system's control, U needs to be optimized to solve the planning problem[3] Planning problem set Ξ is a set of all planning problems.

[3]The notion of U subsumes the cases where the objective of a planning problem is to reach explicit goal states; this could be done simply by assigning a large utility value to those desired states.

Generally, self-adaptive systems face a variety of planning problems (i.e., adaptation scenarios). Although often such scenarios are out of the system's control (e.g., a sudden spike in online traffic demanding an urgent response), systems have a degree of control in selecting the planning problems they solve. For instance, the system may choose its lookahead horizon: should it consider a future of one minute or one hour ahead of the current moment? Some approaches deal with this specific problem of choosing the lookahead horizon [21]. We formalize such unknowns as a more general problem in Section IV-D, and the planning problem set contains all revelant problems, each representing a particular set of inputs to a planner.

Definition 9 (Planner)**.** A planner is defined as $\rho : \Xi \to \Pi$, which is a function that solves planning problem ξ and produces plan π. Given a planning problem, a planner set P is a set of planners that can solve this probiem.

In a particular adaptation scenario, a self-adaptive system has variation in what planner to use for the problem. In practice, planners are affected by their configuration and the format of input. As a result, some planners may not be applicable to some problems. Nevertheless, many planners can solve the same planning problem, thus creating flexibility in what planner to use. For instance, in the context of the self-adaptive cloud setting, researchers have demonstrated the potential of various decision-making approaches such as case-based reasoning [12], automated planning [17], and reinforcement learning [18]. Moreover, for each approach, numerous customizations are possible, which we formalize as individual planners in the planners set. We focus on the interchangeability of the planners, and assume that P is a set of planners that can solve the planning problem in question. This simplification is made without loss of generality, since we always evaluate planners with respect to a certain planning problem (see Sections IV-C and IV-D).

Definition 10 (Utility of planners)**.** The utility of planner is defined as $U : P \times \Xi \to \mathbb{R}$, which is a function that, given planner ρ and planning problem ξ that it is solving, returns a real number indicating the performance of the plan generated by ρ after it has been executed on ξ. This utility function is defined by the utility function for plans: $U(\rho, \xi) \overset{\text{def}}{=} U(\rho(\xi))$.

Definition 11 (Utility of planning problems)**.** The utility of a planning problem is defined as $U : \Xi \times P \to \mathbb{R}$, which is a function that, given set of planners P, maps planning problem ξ to the maximum utility among planners in the set. Formally:

$$U(\xi, P) \triangleq \arg\max_{\rho \in P} U(\rho, \xi).$$

Let us illustrate the given definitions by mapping them back to the self-adaptive cloud setting described in Section II. Planning problems, such as reacting to low response time and seizing an opportunity to reduce operating cost, belong to the planning problem set (Ξ). For these problems, the state space S would consist of both system and environment states; set of actions A would contain adaptation actions such as adding/removing servers; the goal of the system is to maximize utility U. For executions e, $U(e)$ is calculated a sum of instantaneous utilities of individual states $U(s)$. The utility of each state depends on state attributes:response time and operating cost of servers.

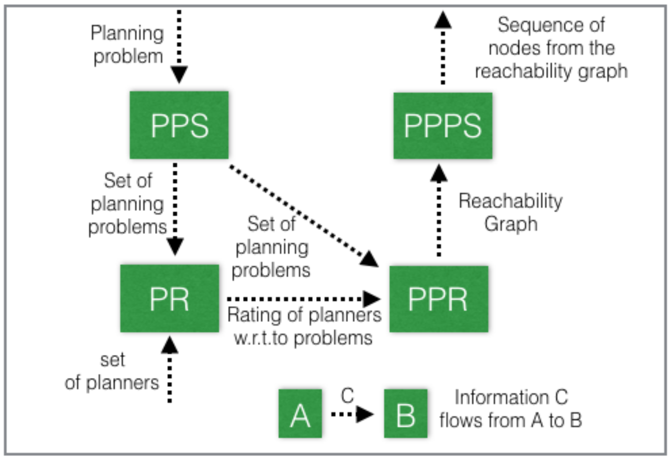
Planning problem
PPS
Set of planning problems
Set of planning problems
PR
set of planners
Rating of planners w.r.t.to problems
PPR
Reachability Graph
PPPS
Sequence of nodes from the reachability graph
A
C
B
Information C flows from A to B

Fig. 1. Decomposition of the hybrid planning problem.

To solve the hybrid planning problem for the cloud planning problem, one may employ several planners, such as MDP and Rainbow—both contained in P. To solve the hybrid planning problem, the self-adaptive system will need to find the best combinations of these two planners by selecting appropriate planning problems (Ξ) and assigning them to planners in advance (to account for their planning delays). In the remainder of the paper we shed light on the subproblems that such a self-adaptive system would need to solve.

IV. Decomposition of the Hybrid Planning Problem

First, let us define the notion of a hybrid plan and planning.

Definition 12 (Hybrid plan)**.** *Hybrid plan* ω is a sequence of plans π_i, each produced by planner ρ_i on planning problem ξ_i:

$$\omega \overset{\text{def}}{=} \langle \pi_1, \pi_2, \dots \rangle, \text{ where } \pi_i = \rho_i(\xi_i).$$

Definition 13 (*HPP*)**.** The *Hybrid Planning Problem* (*HPP*) is, given a planning problem ξ and a set of planners P, produces a hybrid plan sequence of plans from various planners that maximizes the utility.

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

1) *Problem-Planner Path Selection* (*PPPS*): what is the path of problem-planners whose execution yields the maximum utility?
2) *Problem-Planner Reachability* (*PPR*): what is the reachability relation between pairs of problems and planners?
3) *Planner Rating* (*PR*): what quality and timeliness does each planner provide on a given planning problem?
4) *Planning Problem Selection* (*PPS*): what planning problems to solve?

We would like to stress again that these theoretical subproblems of hybrid planning do not prescribe a particular framework or an algorithm for doing hybrid planning. In the remainder of this section, we define and discuss each problem separately. We work our way back from the definition of *HPP*.

A. Problem-Planner Path Selection

The Problem-Planner Path Selection (*PPPS*) problem decides, informally, what sequence of plans from different planners yields the highest utility, and creates a hybrid plan from this composition. To get a sequence of plans for solving *HPP*, a sequence of planner invocations that generates those plans is needed. However, by Definition 10, a plan generated by a planner depends on the planning problem solved by that planner. Therefore, *PPPS* needs to reason about sequences of problem-planner pairs. We first formalize a structure that describes such sequences and serves as an input to *PPPS*.

Definition 14 (Reachability graph)**.** A *problem-planner 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), where deadline d (defined in Section IV-C) is the worst-case time instant when ρ needs to be invoked on ξ.
- $\mathcal{E} \subseteq V \times V$ is a set of edges between pairs of nodes, where each edge ϵ denotes reachability from the first node to the second node.
- $V^i \subseteq V$ is a subset of initial nodes of the graph, linked to variations of the initial planning problems.

An edge (v_1, v_2) in Γ indicates that the plan obtained by executing the planner from v_1 on its planning problem reaches the initial state of the planning problem of node v_2. Through that plan, an edge can be mapped to an execution of the system. Therefore, paths through Γ (which are sequences of edges) mimic executions of the system, guided by concatenated sequences of plans (equivalent to a hybrid plan).

The utility of paths through Γ builds upon the edge utilities in the same way as utility of executions builds upon utilities of states in Definitions 2 and 4. Paths have different utility, and the goal of *PPPS* is to pick a path with the highest utility.

To enable comparison of paths via the ground-truth utility, our next step is to formalize the mapping from edges/paths in Γ to executions. 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 planning problem of the second node: $\mathcal{L}(\epsilon) = \mathcal{L}(v_1.\rho(v_1.\xi))^\frown\langle v_2.\xi.s^i\rangle$. Path (i.e., sequence of edges) $\langle \epsilon_1, \dots, \epsilon_n\rangle$ maps to its execution composed of constituent edges' executions: $\mathcal{L}(\langle \epsilon_1, \dots, \epsilon_n\rangle) = \langle \mathcal{L}(\epsilon_1), \dots, \mathcal{L}(\epsilon_n)\rangle$. This mapping lets us define utility of edges and paths.

Definition 15 (Utility of edges and paths)**.** The utility of edge ϵ is a function $U : \mathcal{E} \to \mathbb{R}$ that maps ϵ to the utility of the edge's execution. Formally, $U(\epsilon) \overset{\text{def}}{=} U(\mathcal{L}(\epsilon))$. Similarly, the utility of a path (a sequence of edges) $\kappa = \langle \epsilon_1, \dots, \epsilon_n\rangle$ is a function $U : \mathcal{E}^n \to \mathbb{R}$ that is defined as $U(\kappa) \overset{\text{def}}{=} U(\mathcal{L}(\kappa))$.

With these definitions, we are ready to fully formalize *PPPS*.

Definition 16 (*PPPS*)**.** The *Problem-Planner Path Selection* (*PPPS*) problem is, given reachability graph Γ, find a sequence of edges starting from any of its initial nodes V^i that maximizes the utility:

$$\mathit{PPPS}(\Gamma) \overset{\text{def}}{=} \arg\max_{\kappa \in \mathcal{E}^n} U(\kappa)$$

In practice, *PPPS* would be the last algorithmic step made in planning, before the hybrid plan is handed over to the execution component. Consider a rapid spike of online traffic in a cloud-based system from Section II. *PPPS* can compare two possible paths: activate fast planner and stick to it, or activate a fast planner but switch to slow planner after the spike. Since the slow planner yields larger utility on average, *PPPS* would pick switching to it after an urgent situation has cleared, thus improving the system's utility.

B. Problem-Planner Reachability

The purpose of the *Problem-Planner Reachability* (*PPR*) problem is to build reachability graph Γ used by *PPPS*. To build this graph, *PPR* relies on two inputs: planning problems Ξ that need to be solved (from *PPS*) and available planners P that were rated in utility (U) and invocation deadline (d) on each ξ (from *PR*).

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

Constructing edges of Γ is more complex. For each pair of nodes v_1 and v_2, we need to determine whether there is an edge between them. The edge exists if and only if two conditions are met:

1) *Preemption:* after executing the plan from v_1, the system should reach to initial state of the planning problem in v_2. Only in this case the plan for node 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 at least its worst-case planning time units earlier than the execution of the system began. Although estimating planning time is a significant obstacle in implementing practical solutions to *PPR*, mathematically there is only one reason for a time early enough not to be found—when it is before $t = 0$. Therefore, the condition for ρ having enough time before its execution is: $d(\rho) > \sum_{\epsilon_i \in \langle \epsilon_0, \dots \epsilon_n\rangle} time(\mathcal{L}(\epsilon_i))$, where function $time$ returns the duration of execution corresponding to an edge.

Now that we have fully defined construction of graphs and edges, we are ready to define *PPR* formally.

Definition 17 (*PPR*)**.** The *Problem-Planner Reachability* (*PPR*) problem is, given planning problems Ξ, planners P, and utility U and deadline d functions, to find reachability graph Γ with edges satisfying the preemption and timing conditions (see above).

In practice, *PPR* 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 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 Rating

The purpose of the *Planner Rating* (*PR*) problem is, given a particular planning problem ξ and a set of planners P, is to rate the performance of these planners on that problem. These ratings are an essential part of PPR (Section IV-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 defined in Section III. For timeliness, we adopt the worst-case model of time: we assume the knowledge of the maximum time needed by a particular planner for a particular planning problem. Thus, *PR* requires finding the worst-case planning time for each planner.

Since planners are functions, their output plan is fully defined by a planning problem. Once executed, the plan's utility is determined as well. Therefore, the only necessary inputs to *PR* are planning problem ξ and space of planners P. Therefore, the quality and timeliness are determined by the execution of that plan.

PR has two outputs:

1) Utility $U(\rho)$ for each $\rho \in P$ that is the utility accrued by the execution of the planner's output on the given planning problem, $U(\rho) = U(\mathcal{L}(\rho(\xi)))$.
2) Deadline $d : P \times \Xi \to \mathbb{R}$ is a function of a planner that returns a real number that indicates the worst-case delay between starting a planner (on a certain problem) and receiving its plan.

Definition 18 (*PR*)**.** The *Planner Rating* (*PR*) *problem* is, given planning problem ξ and set of planners P, to find the utility U (ρ) and deadline d (ρ) of each planner $\rho \in P$ on ξ.

To solve *PR* in practice, one would need to create algorithms to measure utilities and deadlines for planners. To the authors' knowledge, the absolute majority of existing planner implementations do not provide up-front guarantees on either of these two characteristics. Therefore, two general approaches are possible: (i) designing new planners with guarantees of quality and timeliness on given planning problems, and (ii) approximately predicting 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 *PR* explains how to evaluate such predictions of planners' quality in a uniform way.

D. Planning Problem Selection

The goal of the Planning Problem Selection (*PPS*) problem is to decide what planning problems should be solved. *PPS* sets the direction of planning, which in turn sets the adaptation course.

Formally, at every moment, an infinite number of planning problems can be formulated and solved. According to Definition 8, one can arbitrarily select the initial state, the subset of actions, the subset of the state space, and the utility function. If the utility function is fixed, the number of problems is finite, but still much larger than feasible to formulate and solve in practice. Hence, *PPS* reduces the space of all possible planning problems (Ξ) to a smaller set of planning problems that is input to *PPR* and *PPPS*.

As time passes, the set of relevant planning problems changes. For example, some initial states become not reachable, and these problems become obsolete. At every moment of time, there is a set of relevant planning problems Ξ. In Ξ, there is a subset of problems that yield the largest utility—if solved and the plan is executed immiediately. Due to the delay in planning, we must consider problems in the future, the solution of which needs to be started in advance, so that the plan is executed just-in-time. Therefore, in a general case, *PPS* needs to return multiple planning problems—not a single one with the highest utility.

So far we treated planning problems as timeless objects, with time implicitly encoded in s^i. Within the discussion of *PPS* we make time explicit for planning problems. However, this extension does not affect other subproblems of *HPP*, since time can be discarded when transferred to these subproblems.

Definition 19 (Timed Ξ)**.** *Time-specific space of planning problems*, Ξ_t, is the set of planning problems $\xi \in \Xi$ that have the initial state $\xi.s^i$ with time t.

The inputs *PPS* are the time-specific space of planning problems Ξ_t and the initial problem ξ^i. Problem ξ^i is the initial planning problem given to hybrid planning. This problem contains the utility function and a description of all reachable states and actions. It also contains the original system state, from which execution will start.

The output of *PPS* is a set of planning problems $\Xi_t^* \subseteq \Xi_t$ that will solved, and their plans will inform the behavior of the self-adaptive system. As discussed earlier in this section, *PR* will rate planners on Ξ_t^*, *PPR* will analyze reachability between problem-planner pairs, and finally *PPPS* will select the best sequence of plans and, consequently, planner executions.

The goal of *PPS* is to select planning problems with maximum utility (as described in Definition 11). Since the space Ξ changes every moment, *PPS* should produce problems with maximum instant utility. That is, the best plans that the planning problems produce (when solved by their best planners, as decided in *PR*) should yield maximum utilities if executed at the same time where the problem originates (i.e., its s^i has that time).

Definition 20 (*PPS*)**.** The *Planning Problem Selection* (*PPS*) problem is, given the initial planning problem ξ^i and time-specific planning problem space Ξ_t, selects the best planning problem, $\arg\max_{\xi \in \Xi_t} U(\xi)$, for each time t.

For illustration of *PPS*, imagine a scenario—the block world example in [20]. It is a static planning problem: the space of planning problems does not evolve with time (although it can be infinite). Therefore, a fixed non-empty set of planning problems yields maximum utility. This set would be the output of *PPS*. If any problem in this set is selected and solved, the resulting plan to achieve maximum utility indefinitely (assuming no uncertainty in actuation or environment). Such a plan is guaranteed to be the optimal plan, by the Definition 11.

Two obstacles make practical implementation of *PPS* difficult. First, it is impossible to decide the best planning problem for each time moment. Therefore, self-adaptive systems would need a mechanism to decide what moments to give to *PPS*. This can be done periodically or based on various heuristics. For example, when the incoming traffic is relatively stable in the cloud system, the periods of *PPS* can be larger. The *PPS* triggering mechanism will be affected by the loss of expected utility by picking planning problems in the future. This loss will occur due to lower-quality predictions of the environment behavior. Implementations of hybrid planning will need to estimate this loss and account for it.

The second substantial practical issue for *PPS* is the mutual dependency between *PPS* and *PR*: a problem cannot be evaluated without planners, and planners—without a problem. There does not seem to be a theoretical way to break this circularity. Therefore, systems will need to employ approximate solutions. One example of such solution is to use machine learning to predict which problems and planners would yield the most utility. Specifically, one would identify two independent sets of features that predict, respectively, the utility of problems and planners.

V. Discussion

Here we discuss two characteristics of the proposed hybrid planning formalization: its use for experimentally validating 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 pairs of problem-planners. We use utility to directly measure "goodness" of decisions in subproblems. We use reachability to combine plans into a hybrid plan with guaranteed preemption and timing. While these idealizations are not directly implementable, they do provide a single uniform way to evaluate future solutions to the subproblems of hybrid planning.

Utility and reachability enable a workflow of evaluation:

- Implement a hybrid planner and a simulation of a system.
- Execute the planner on the system in several scenarios, logging complete execution traces.
- Calculate utility of traces according to Definition 4.
- Reconstruct a reachability graph for each scenario.
- Perform what-if simulations to determine:
  - Are there more optimal paths?
  - Are nodes locally optimal in their neighborhood?
  - Does removing nodes improve the performance?
  - Are some edges incorrect or missing?
- The identified improvements indicate the delta between the empirical utility and theoretical utility.

This is a repeatable, robust evaluation procedure for hybrid planners, grounded in theoretical concepts. In fact, it is also applicable to prior work on combining different contingency plans [19]. Although such experiments can be computationally expensive, they yield valuable insights into the behavior of hybrid planners and opportunities for their improvements.

B. Generality

A careful reader might ask: is this the right formalization and does it apply to all possible hybrid planners? We believe that several such formalizations are theoretically possible, and to our best knowledge this is the only one in existence. The distinctive feature of this formalization is *parsimony*—our foundation uses only essential concepts broadly applicable to planners, and we introduce the least restrictive assumptions that still allow precise formal definitions. Below we explicitly summarize our assumptions behind and assess the scope of their validity.

Known utility of states/executions: this assumption holds in the majority of contexts where self-adaptation is applied to software systems. Exceptions are cases when experimental data remains incomplete or inaccessible. One example is sophisticated cyber-physical systems where physical state may be volatile and difficult to log in its entirety.

Instantaneous solutions to subproblems: our formalization factors in only the delays of planning itself, but not delays of solving *PPPS*, *PPR*, *PR*, and *PPS*. This assumption holds if solving these problems takes negligible time compared to the time scale of planning and execution, or if the solutions can be precomputed offline.

Fixed space of planners: the formalization is inspired by realistic contexts where planning tools are known before they are executed in a system. This assumption is generally valid in practically all contexts where planners are used today.

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 the time assumption opens a promising direction of future work—predictable planners in self-adaptation. An alternative research direction is to model the expected planning time as probabilistic. Although this assumption is more realistic, it would propagate probabilities throughout the formalization, making reasoning too approximate to define a precise, deterministic theoretical baseline for hybrid planning.

To summarize, this paper takes on a sophisticated problem of hybrid planning and decomposes it into four computational subproblems. Due to formalization, the definitions from the paper may serve as a validation framework for practical solutions to hybrid planning. We expect that, due to complexity in hybrid planning, hybrid planning frameworks will greatly vary in formal approaches, design patterns, and components that address the subproblems. This paper provides a unifying way to experimentally evaluate such frameworks, based on formal notions of utility and reachability.

We would like to encourage the self-adaptive systems community to actively participate in contributing to engineering solutions to the subproblems of hybrid planning. According to prior work [16] [19], hybrid planning is a promising way to both improve self-adaptation and combine multiple self-adaptive frameworks, thus increasing the potential for industrial adoption. However, the openness and complexity of hybrid planning creates the possibility for many diverse approaches. Therefore, extensive experimentation is needed to provide efficient, usable, and general approaches to planning in 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] B. Nebel, The FF Planning System: Fast Plan Generation Through Heuristic Search, in: Journal of Artificial Intelligence Research, Volume 14, 2001, Pages 253 - 302.

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

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

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

[5] 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.

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

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

[8] 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.

[9] C. Klein, M. Maggio, K.-E . Irz̀en, 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.

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

[11] 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.

[12] 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

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

[14] Zack Coker, David Garlan, Claire Le Goues: SASS: Self-Adaptation Using Stochastic Search. SEAMS@ICSE 2015: 168-174

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

[16] A. Pandey, G. A. Moreno, J. Càmara, and D. Garlan, 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.

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

[18] Barry Porter, 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

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

[20] Gupta, N.; Nau, D. "On the Complexity of Blocks-World Planning" (PDF). Artificial Intelligence. 56: 223-254

[21] Roykrong Sukkerd, Javier Cam̀ara, David Garlan, Reid G. Simmons: Multiscale time abstractions for long-range planning under uncertainty. SEsCPS@ICSE 2016