Hybrid Planning with Receding Horizon: A Case for Meta-self-awareness

Sona Ghahremani and Holger Giese
Hasso Plattner Institute, University of Potsdam, Potsdam, Germany, Email: {firstname.lastname}@hpi.de

arXiv:2108.11170v1 [cs.SE] 25 Aug 2021

***Abstract*—The trade-off between the quality and timeliness of adaptation is a multi-faceted challenge in engineering self-adaptive systems. Obtaining adaptation plans that fulfill system objectives with high utility and in a timely manner is the holy grail, however, as recent research revealed, it is not trivial. *Hybrid planning* is concerned with resolving the time and quality trade-off via dynamically combining multiple planners that individually aim to perform either timely or with high quality. The choice of the most fitting planner is steered based on assessments of runtime information. A hybrid planner for a self-adaptive system requires (i) a decision-making mechanism that utilizes (ii) system-level as well as (iii) feedback control-level information at runtime.**

In this paper, we present HYPEZON, a hybrid planner for self-adaptive systems. Inspired by model predictive control, HYPEZON leverages receding horizon control to utilize runtime information during its decision-making. Moreover, we propose to engineer HYPEZON for self-adaptive systems via two alternative designs that conform to *meta-self-aware* architectures. Meta-self-awareness allows for obtaining knowledge and reasoning about own awareness via adding a higher-level reasoning entity. HYPEZON aims to address the problem of hybrid planning by considering it as a case for meta-self-awareness.

***Index Terms*—meta-self-awareness, self-adaptive systems, hybrid planning, receding horizon control, coordinating, model predictive control**

I. INTRODUCTION

Rapidly changing requirements, highly dynamic environments, and unpredictable operating conditions demand for runtime adaptation of software systems. Providing timely and high quality adaptation plans is the ultimate goal of an adaptation manager. However, constructing a single automated adaptation policy that satisfies both of these conflicting requirements is challenging [12]. *Proactive* optimization-based policies often require an exhaustive search in the possible adaptation space which renders attaining optimal adaptation plans time-intensive [27]. Additionally, while *reactive* condition-based solutions for adaptation deliver adaptation plans timely, they often fail to find the optimal solutions [10].

Hybrid planning employs control mechanisms where multiple adaptation policies are orchestrated to jointly carry out timely and optimal adaptations [5], [25]. A hybrid planner for a self-adaptive system (SAS) implements the ability to reason about different available adaptation policies, varying in the quality and timeliness of their plans, based on the runtime conditions. Time-critical operation conditions that demand timely rather than optimal adaptations benefit from employing cost-effective adaptation policies with short planning time. Under less time-sensitive operation conditions, optimization-based policies may provide high quality adaptation plans.

While Control Theory has established mathematically grounded and practical frameworks for managing complex systems [1], [13], it restricts the scope of the controllers to calculating set-points and prescribing required changes in the system input parameters [12]. The black-box-oriented scheme of the Control Theory further extends towards *adaptive control*, where the controller may change its own control regime [22]. This requires the controllers to have adjustable parameters. In the realm of self-adaptive systems, adaptive control is perceived as reasoning about the adaptation logic [26]. The reasoning requires observing the behavior of the control loop in terms of effectiveness and performance, realizing the need for change, and prescribing the necessary decisions to steer the controller towards the desired behavior. *Meta-self-awareness*, a notion surfacing only recently, captures the requirements for equipping self-aware systems with advanced self-reflective properties [7]. As a result, systems with meta-awareness properties can reason about changing trade-offs during their lifetime [23]. The control design and architecture of a meta-self-aware system builds on Control Theory as a prominent base, however, it extends the involvement *scope* of the higher-level control loops in the lower-level entities.

There exists various research efforts in combining multiple adaptation solutions to fulfill the contradicting trade-offs in SAS [20], [31]. [5] proposes a hybrid approach that combines control theory principles with AI techniques to optimize the adaptation process. [30] presents a hierarchical hybrid planner where a learning-based policy adapts a rule-based policy to improve its decisions. The hybrid planning approach in [24] operates an optimization-based planner in the background while a deterministic policy adapts the system. Before each adaptation, the hybrid planner checks if the optimization-based policy can provide a plan. [32] presents a concurrent approach for self-adaptation in which a model of SAS is synchronized with multiple views on that model and each partial model has its own adaptation mechanism. Condition-based rules are used in [4], [21] to construct reactive hybrid planners. The main focus in providing hybrid solution for SAS so far has been set on developing individual adaptation policies such that, in coordination together, they cover a large spectrum of the solution space for SAS. In this paper, we focus on the orchestrating entity, i.e., the realization of the decision-making mechanism within the hybrid planner. Our proposed solution has the characteristics of a generic hybrid planner since it considers the employed adaptation policies as a black-box, thus, can coordinate arbitrary adaptation policies.

We address the problem of hybrid planning by considering it as a case for meta-self-awareness. We present HYPEZON, a hybrid planner for self-adaptive systems. HYPEZON implements the planning phase as a controller conforming to the scheme of *model predictive control*, thus, leverages *receding horizon* to utilize runtime information and adjust its control parameters at runtime. To engineer HYPEZON for SAS, we propose two alternative designs, *external* and *internal*, that conform to *meta-self-aware* architectures. The designs build on the framework for realizing meta-self-awareness in the architecture from our earlier work [19].

We study the effectiveness of the designs for hybrid planning by answering the following Research Questions (RQ). **RQ1** how do internal and external designs for meta-self-awareness affect HYPEZON? **RQ2** how does HYPEZON perform in comparison to a deterministic hybrid planner? **RQ3** what are the effects of hybrid planning on the quality and timeliness of the adaptation? We show that meta-awareness capabilities, realized either by the external or internal design, are beneficial for hybrid planning as they provide extended control flexibility at runtime.

Section II discusses the prerequisites. A motivating example is presented in Section III. Section IV presents HYPEZON and Section V presents the design and application of HYPEZON as a case for meta-self-awareness. The **RQ**s are investigated in Section VI and Section VII concludes the paper.

II. PREREQUISITES

A. Self-adaptive Systems

The execution of an *adaptation action* $a \in A$ with A the set of available actions, results in adapting the SAS from state s to s' with s and $s' \in S$ and S the *state space* of the SAS. A *policy* is generally defined as a set of control decisions that map states to actions [27]. An *adaptation policy* π represents an encapsulation of the system's adaptive behavior governing the choice of adaptation actions when applicable. For each state s, $\pi(s)$ indicates the adaptation action a to be executed, i.e., $\pi(s) = a$. $EU(s)$ represents the expected utility of s i.e., a scalar value as a quality metric that identifies the degree to which system goals and requirements are satisfied in s. For all the applicable adaptation actions $a \in A$ in state s, an *optimal* $\pi(s)$ chooses the action that maximizes the expected utility of the subsequent state s', i.e., $EU(s')$.

The utility of a system can be defined as a function of the system *quality dimensions*. A common practice to obtain a representative utility function is via the system's Service Level Agreements (SLA's) where the business preferences define the system *objectives*. A multi-objective utility function is an aggregation of multiple quality dimensions where each dimension represents a business objective. *Utility sub-functions* are employed to assign values to the dimensions [33].

A *static* adaptation policy maps an action a to state s based on design-time estimates for the resulting state s'. Therefore, the estimations for $EU(s')$ are agnostic to using any runtime observation. In contrast, a *dynamic* adaptation policy leverages runtime observations, available during system execution, to *compute* expected utility values for the applicable actions.

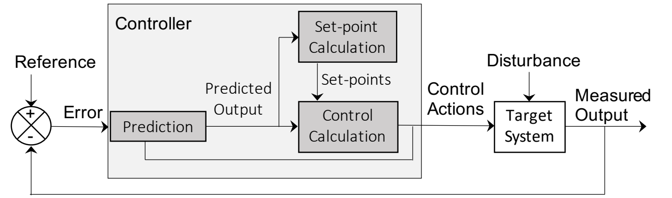

Fig. 1. Block diagram of a feedback control system. The gray box extends the controller to illustrate a model predictive controller.

B. Hybrid Planning for SAS

Hybrid planning for SAS in general refers to combing two or more adaptation policies to adapt the system. More specifically, a *Coordinating Hybrid Planner* (CHP) combines adaptation policies that (i) target the same SAS and (ii) the same *planning problem* but (iii) keeps their planning phases separate. The planning phase of a SAS employing a CHP can be subdivided into the original planning phases from the combined policies [31]. An *adaptation issue* refers to a deviation from the desired state and indicates that the system, in its current state s, requires an adaptation. A *planning problem* is a set of adaptation issues that simultaneously affect the SAS. For a planning problem, a *plan* is an ordered list of adaptation actions that each action resolves at least one of the adaptation issues in the planning problem. The *look-ahead horizon* is the time steps into the future that are considered during planning. A *planning horizon* Φ is a prefix of the look-ahead horizon that is planned for. A planning horizon of size $|\Phi| \leq l$ with l the size of the look-ahead horizon, only plans for $|\Phi|$ out of l adaptation issues in the look-ahead horizon. An infinite planning horizon allows for considering the entire look-ahead horizon for planning. The *utility of plan* is the expected utility of the target system state after the execution of the adaptation actions constituting the *plan*.

A CHP is concerned with two aspects of the available policies: quality and timeliness. For a given planning problem, the quality of a policy π_i is quantified as the expected utility of the plan provided by π_i, i.e., $EU(plan_i)$. For timeliness, an estimation of the time required by π_i to provide a plan, i.e., $ET(plan_i)$, is required [25]. Such estimations can be obtained via theoretical modeling such as employing worst-case time models [24] or based on empirical profiling.

C. Feedback Control for SAS

A control feedback loop that governs a SAS observes the system output in *sampling intervals* $\mathcal{I}$ and adapts the system towards its *set-points* to prevent violation of the system requirements and goals over a *prediction horizon*. The *target system*, i.e., SAS is controlled via the *control actions* from the *controller*–see Fig. 1. The *reference* input is the desired value of the system's *measured output*. The goal is that, despite the *disturbance* affecting the target system, the measured output is sufficiently close to the reference. For this purpose, the difference between the measurement and the reference, i.e.,

the *error* is fed back to the controller to determine the control actions required to achieve the reference value.

Model Predictive Control (MPC) is a technique that formulates a multi-variable optimization function, e.g., a utility function, to generate *set-point*s. Set points define the target values for *control calculations*. Control calculations determine a sequence of M control actions, i.e., *control horizon*, such that the *predicted output* moves towards the set-points over a finite *prediction horizon* P [18]. The number of the predictions P is referred to as the *prediction horizon* while the number of the control actions M is called *control horizon*. The MPC *receding horizon control* suggests that although a sequence of M control actions is calculated at each time point, only the *first* action is executed. A new sequence is calculated after new observations become available. Employing a receding horizon of size *one* supports the case where the variables available for the control calculations change from one execution time to the next. If the control structure changes from one control execution time to another, but the MPC controller does not recalculate the parameters, the subsequent control calculations may become *ill-conditioned* [29]. For a SAS, before each adaptation, the adaptation policy π optimizes an objective function to select the adaptation actions that maximize the said function. MPC has been proven effective in formulating the optimization problem where the control actions represent adaptation actions and the targets are the system states with maximum EU [2].

D. Self-awareness and Meta-self-awareness

A *self-aware* system is identified by two main characteristics. First, the ability to *learn* models capturing knowledge about the system, its context, and its goals on an ongoing basis. Second, *reasoning* based on the models for analysis and planning concerns. Computational self-awareness is achieved via a Model-based Learning, Reasoning, and Acting loop (LRA-M loop). There can be multiple variations to Acting, e.g., explaining, reporting, suggesting, and adapting. Self-adaptation, realized via a MAPE-K feedback loop, is one of the advanced characteristic of the self-aware systems where the *scope* of *Acting* is set to *Adapting* [23]. As a result, a system realizing MAPE-K loop becomes aware of itself and its context. The *object* of the awareness is the entity being reasoned upon. The *subject* of the awareness is the entity performing the reasoning. In the following, we target self-aware systems with adaptation capabilities.

A *meta-self-aware* system can obtain knowledge about its own awareness and how it is exercised. A higher-level self-aware entity, e.g., a MAPE-K control loop, *reflects* on the benefits and costs of maintaining increased awareness as well as the capacities for it [7]. Meta-self-awareness is concerned with two classes of objects; The elements of the lower-level awareness loop, e.g., a learning process or an adaptation logic, and the output of these elements, i.e., the models or specifications produced by the object being reasoned upon [23]. In order to explicitly capture the meta-self-awareness properties in the architectural design, meta-self-awareness can be realized

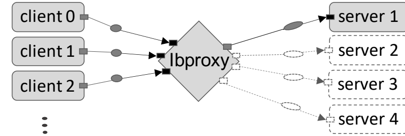

Fig. 2. `Znn.com` architecture. Dashed lines represent available but inactive elements.

either as a built-in capability or as an external meta-awareness layer [19]–similar to the internal and external approaches for engineering systems with self-adaptation properties [9].

III. Motivating Example

As a running example we employ `Znn.com` [8] that simulates a news service website. Clients make content requests to one of the servers. A load balancer distributes requests across a server pool. The size of the pool can be dynamically adjusted to balance the server utilization against the service response time. `Znn.com` is a web-based client-server system conforming to an N-tier style–see Fig. 2. Certain system and client information such as server load, request response time, and the connection bandwidth can be monitored. The goal of `Znn.com` is to provide short response times to the clients while keeping the cost of the server pool within the budget. Four types of adaptation issues may affect `Znn.com`: `latency` in the response times, `overBudget` cost of adding servers to the pool, `underUtilized` servers, and `lowQuality` contents. There are four adaptation actions applicable to modify the configuration of `Znn.com`: `Discharge-server`, `Enlist-server`, `Increase-quality`, and `Reduce-quality` of the content. Response time, content quality, server utilization, and budget are the four objectives of `Znn.com`. These objectives are captured by four quality dimensions respectively. Each dimension is represented by a utility sub-function as described in Table I–see Section II-A.

RT in Table I is the estimated client response time and RT_{max} is set to 90 seconds, that is when `Znn.com` throws a *request timeout* exception and ends the session. A server in `Znn.com` can transfer content with three different qualities that are quantified as: (low, 0), (medium, 0.5), and (high, 1). Therefore, $Server.quality$ can have one of the 0, 0.5, or 1 values. $Server.utilization$ is the percentage of the server capacity that is in-use, and finally, $Server.cost$ is the operational cost of the server which can vary for different providers.

$$U_{znn}(s) = \sum_{client} w_r u_R + \sum_{server} (w_q u_Q + w_u u_U - w_c u_C) \quad (1)$$

Similarly to [8], we define the overall utility of the system as a weighted sum of the utility sub-functions. The weights w_i in Table I are extracted from [8]. For each state s, the overall utility of `Znn.com` is defined according to (1). For each target

TABLE I
Utility Sub-functions for `Znn.com`

ID	Quality Dimension	Utility Sub-function	w_i
u_R	Response time	$u_R = 1 - \frac{RT}{RT_{max}}$	0.4
u_Q	Content quality	$u_Q = Server.quality$	0.2
u_U	Server utilization	$u_U = Server.utilization$	0.1
u_C	Cost	$u_C = Server.cost$	0.3

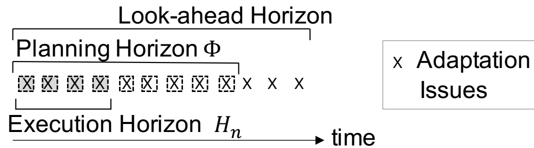

Fig. 3. Look-ahead, planning and execution horizon in HYPEZON.

state s', the expected utility of the state is an estimation of (1), i.e., $EU(s') = \hat{U}_{znn}(s')$.

IV. HYBRID PLANNING WITH RECEDING HORIZON

In this section, we present HYPEZON, a coordinating HYbrid PlannEr for SAS employing receding horiZON control. HYPEZON aims to address the planning problem in a SAS at runtime by considering multiple adaptation policies and selecting based on the time and quality objectives. HYPEZON implements the planning phase of a MAPE-K loop. During an adaptation cycle, once the analysis phase has detected the need to plan an adaptation, i.e., there exists a planning problem–see Section II-B–HYPEZON takes the system state, the set of available adaptation polices, and estimations of $EU(plan)$ and $ET(plan)$ for the available policies. Then, the planner decides which policy best suits the current operation condition.

HYPEZON implements the planning phase as a controller conforming to the scheme of model predictive control with receding horizon. The MPC-based planner, at each sampling instance $\mathcal{I}$, makes new measurements, and based on the operation condition, decides if a policy switch is required. The process of resolving a set of adaptation issues via selecting a set of adaptation actions that maximize a utility function is formulated as an MPC receding horizon control problem. HYPEZON implements a hybrid planner as an MPC controller via mapping the look-ahead and planning horizons of the planner to the prediction and control horizons of an MPC controller respectively–see Section II-C.

HYPEZON extends the notion of MPC receding horizon of size *one* to an *execution horizon* H_n with adjustable size n. Fig. 3 shows an example of a look-ahead horizon, planning horizon, and execution horizon in HYPEZON–see Section II-B. As explained in Section II-C, in MPC, the control horizon is a list of *all* the actions that are planned during an adaptation cycle. This is captured as the planning horizon Φ in HYPEZON. As stated by Pandey et al. [25], it is difficult to verify the compatibility between the plans of different policies, how to choose the planning horizon, and when to stop using one plan and switch to another policy for planning. In the following, we describe how HYPEZON addresses these challenges. HYPEZON uses runtime information such as the planning time of the adaptation policies, system load, number and type of the adaptation issues, and the cost of switching between the policies to decide on the size of the planning and execution horizons. Employing planning and execution horizons with adjustable size provides for runtime flexibility. Before choosing a policy, HYPEZON adjusts the size of the planning horizon Φ with respect to its estimation of the policy planning time, i.e., $ET(plan)$. For example, if `Znn.com` is affected by large numbers of `latency` issues as well as `lowQuality` issues, HYPEZON may restrict the planning horizon to first consider the more critical issues, i.e., `latency`. This way, HYPEZON reduces the expected planning time by reducing the size of Φ, thus, the `latency` issues are resolved relatively faster.

An execution horizon H_n only considers the first n adaptation actions in the planning horizon for execution in the current adaptation cycle. After executing the n adaptation actions, HYPEZON stops the execution of the plan and the remaining unresolved issues are considered together with the newly observed issues in a subsequent adaptation cycle as a new planning problem. Employing execution horizon of small size in HYPEZON results in utilizing the most recent adaptation issues immediately. In contrast, large execution horizons ignore the recent observations until all the actions in the planning horizon are executed–see Fig. 3. Small sizes for H_n demand more frequent planning. Moreover, the execution of the actions that are in the planning horizon and not in the execution horizon is postponed to the subsequent adaptation cycle(s). In cases where the planning phase of the adaptation policy has a large overhead, frequent planning might affect the adaptation time negatively. When HYPEZON switches the adaptation policy, the employed policy plans for the remaining adaptation issues whose corresponding adaptation actions in the planning horizon were not included in the execution horizon. Moreover, the employed policy also considers the newly detected adaptation issues. Consequently, after each policy switch, the planning problem is considered anew during the control calculation in HYPEZON. This way, HYPEZON guarantees that after a policy switch, the active policy calculates the plan according to the most recently observed conditions while taking into account the already existing issues.

In order to guarantee the compatibility between a plan and the planning problem, HYPEZON only executes one policy at a time and avoids concurrent executions of multiple policies. As a result, a planning problem that is assigned to a policy remains unchanged during the planning time. This way, once the plan is ready, HYPEZON does not check if the plan is still applicable to the current planning problem. This feature in HYPEZON avoids the runtime overhead that is caused by compatibility analysis between the planners. However, concurrent executions of planners may reduce the time that the hybrid planner has to wait until a plan is ready.

V. HYPEZON: DESIGN AND APPLICATION

In this section, we argue that equipping a SAS with hybrid planning should be realized as a meta-self-awareness property. To this end, building on our previous study towards making meta-self-awareness visible in the architecture [19], we propose two designs to engineer a SAS with meta-self-awareness properties and show how the two designs are realized in HYPEZON. The self-awareness capabilities are realized via MAPE-K loop–see Section II-D.

A. Meta-self-aware Designs Realizing Hybrid Planning

External design In order to explicitly separate the awareness and meta-awareness levels, as depicted in Fig. 4-(a), two MAPE-K loops are employed. The loop at the *meta-awareness*

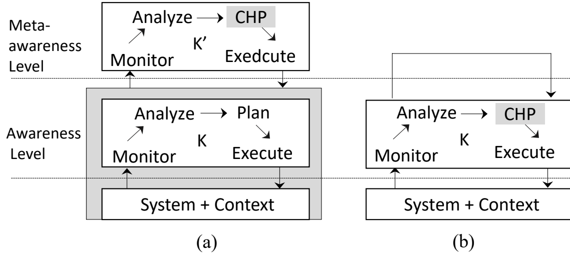

Fig. 4. External (a) and internal (b) design for meta-self-awareness.

level implements the CHP. The higher-level loop observes the awareness level in combination with the system and context, reasons about them, and adapts them accordingly.

The external design operates the CHP on a different timescale than the lower-level MAPE-K loop. The lower-level loop is executed more frequently to guarantee timely adaptation concerning the part of the system under its direct control. The meta-awareness loop however, operates at a relatively larger timescale since it is inherently concerned with relatively more sparse phenomena to react upon [11].

Internal design An internal realization of the meta-self-awareness properties is possible via employing an *awareness self-loop*–see Fig. 4-(b). In this design, the MAPE-K loop *observes* and *affects* itself. The subject and object of meta-awareness are not architecturally separated, consequently, one element, i.e., the MAPE-K loop, performs both the *reasoning* and *being reasoned about* parts of the meta-awareness. In the internal design, the awareness level is also aware of itself.

The internal design implements the awareness and the meta-awareness properties in an intertwined manner and keeps the subject of the meta-awareness close to the object. In this design, the meta-awareness self-loop, hence the CHP, operates at the same timescale as the awareness loop.

In the proposed designs, the focus is on the *functional* aspects enabled by each design rather than the *architectural* aspects. However, the architectural decision of separating the awareness and the meta-awareness loops in the external design and combining them in the internal design steers the decisions relevant to the functional aspect, i.e., the sampling and execution intervals of the CHP in each design.

B. Application

HYPEZON is realized as a meta-awareness subject via both external and internal designs. The variants are called HZ$_e$ and HZ$_i$ respectively. The external design, thus, HZ$_e$ operates the meta-awareness loop in coarser time intervals compared to the awareness loop. Consequently, compared to HZ$_i$, the MPC controller in HZ$_e$ has larger sampling intervals ($\mathcal{I}$)–see Section II-B. Algorithm 1 shows a high-level description of HYPEZON.

HYPEZON is concerned with (i) *control parameter tuning*, i.e., tuning the size of the planning and execution horizons, and (ii) *policy switch*. As depicted in Algorithm 1, HYPEZON requires system state s and the set of available adaptation polices Π. Π also includes estimations of $EU(plan)$ and $ET(plan)$ for the available policies. The planning horizon Φ is initially set to fully include the look-ahead horizon, i.e.,

Algorithm 1 Hybrid Planning with HYPEZON

Require: s, Π, $RT_{\mathcal{I}}$
1: $\Phi \leftarrow$ look-ahead horizon
2: **if** $RT_{\mathcal{I}}$ **optimal then**
3: $\pi_{curr} = \pi_i \in \Pi \ni i = \operatorname*{argmax}_i EU(plan_i)$
4: $H_n \leftarrow \infty$
5: **else if** $RT_{\mathcal{I}}$ **inRange then**
6: **for all** $\pi_j \in \Pi$ **do**
7: **if** $c_{curr,j} + ET(plan_j) + RT_{\mathcal{I}}$ **inRange AND** $j = \operatorname*{argmax}_j EU(plan_j)$ **then**
8: $\pi_{curr} = \pi_j$
9: **adjust**(H_n)
10: **if** $\pi_{curr} = null$ **then**
11: **adjust**(Φ)
12: Go to 3
13: **else if** $RT_{\mathcal{I}}$ **high then**
14: $|\Phi| \leftarrow 1$
15: $H_n \leftarrow 1$
16: $\pi_{curr} = \pi_j \in \Pi \ni j = \operatorname*{argmin}_j ET(plan_j)$
17: List of Actions $\leftarrow \pi_{curr}(s)$
return List of Actions , H_n

all the the existing adaptation issues–see Fig. 3. In addition to the information available as system state s, e.g., current system load and utility, HYPEZON also maintains the average response time of the SAS during the sampling interval $\mathcal{I}$, i.e., $RT_{\mathcal{I}}$. Based on the specific business objectives of SAS, HYPEZON defines three ranges for the average response time: *optimal*, i.e., $RT_{\mathcal{I}}$ is below a minimal threshold, *inRange*, i.e., $RT_{\mathcal{I}}$ is within the acceptable range, and *high*, i.e., $RT_{\mathcal{I}}$ is higher than the permitted upper bound. These thresholds are subject to change at runtime to reflect the changing goals and requirements.

Control parameter tuning For a given planning problem, both HYPEZON variants employ the method *adjust*–line 9 and 11 in Algorithm 1–to tune their control parameters at runtime. The method uses estimations of the $EU(plan)$ and $ET(plan)$ for the policies, the current and estimated system load, and the number as well as the type of the adaptation issues.

Policy switch Switching between policies has a cost as it requires deploying specific settings for the new policy, e.g., initializing a constraint solver or loading prediction models. The switch from policy π_i to π_j is charged with a cost c_{ij} that is subtracted from the system utility. HYPEZON reasons about the trade-off between cost and benefit of the switch at runtime–line 2, 5, and 13 in Algorithm 1. If $RT_{\mathcal{I}}$ is *optimal*, HYPEZON switches to the policy with highest expected plan utility and executes the full plan–line 2-4 (index *curr* represents the current choice). If $RT_{\mathcal{I}}$ is *inRange*, HYPEZON searches for a policy with the highest $EU(plan)$ such that the sum of the policy switch cost, $ET(plan)$, and $RT_{\mathcal{I}}$ is still *inRange*–line 5-8. HYPEZON uses $RT_{\mathcal{I}}$ as an estimate for RT during the next interval. In case HYPEZON does not find a match, the size of the planning horizon is reduced until HYPEZON finds a match–line 10-12. Finally, if $RT_{\mathcal{I}}$ is *high*, HYPEZON sets the size of the planning and execution horizons to one and searches for

a policy with the minimum planning time–line 13-16.

In both variants, decisions for tuning the control parameters and policy switch is made based on the average values over a sampling interval $\mathcal{I}$. Therefore, HZ$_e$ makes estimates of the $EU(plan)$, $ET(plan)$, and system load based on the average values of observations over a longer monitoring period compared to HZ$_i$. Thus, HZ$_e$ collects accumulated observations of system executions that can be used to estimate the operation condition during the next interval. HZ$_i$ however, operates more frequently and decides based on the observations over a relatively smaller $\mathcal{I}$.

VI. Evaluation

In this section, we evaluate the application of the HypeZon variants on `Znn.com`. The experiments are designed to answer the three Research Questions. **RQ1** how do internal and external designs for meta-self-awareness affect HypeZon? **RQ2** how does HypeZon perform in comparison to a deterministic hybrid planner? **RQ3** what are the effects of hybrid planning on the quality and timeliness of the adaptation?

A. Case Study and Deterministic Hybrid Planner

Case study The employed case study is `Znn.com` from Section III. The request arrival traces are generated based on the commonly used (e.g., [6]) web traffic logs of FIFA 98 world cup site [3] and are employed as the input traffic for `Znn.com`. We consider three different traces (TRi)–available in [14]. The traces include clients web content requests from the web servers over the course of 24 hours on three different days. The average number of requests per minute is $10,796$. However, the content requests are not uniformly distributed over time, and demonstrate the *slashdot effect*, i.e., sudden and relatively temporary surges in traffic. As a result of the slashdot effect, the response time of the servers increases above the acceptable threshold and causes `latency` issues for the affected clients. The employed adaptation policy π addresses `latency` via `Enlist-server` or `Reduce-quality` actions. The two actions however could cause `overBudget`, `underUtilized`, and `lowQuality` issues which are dealt with once the slashdot effect wears off via `Discharge-server` and `Increase-quality`.

Deterministic hybrid planner We implemented a deterministic coordinating hybrid planner that uses predefined thresholds on quality attributes of interest, e.g., response time, as constraints. The proposed hybrid planner, CHP$_{dtr}$ henceforth, does not support runtime adjustments of its control parameters and considers look-ahead, planning, and execution horizons with deterministic and predefined sizes and, as a result, exhibits smaller planning overhead at runtime. CHP$_{dtr}$ takes current state s, set of available policies Π, current response time RT_{curr}, and response time threshold RT_{thr} as inputs. If RT_{curr} exceeds RT_{thr}, CHP$_{dtr}$ switches to a policy with a smaller $ET(plan)$, otherwise, a more time-intensive policy obtaining higher quality is employed.

B. Experiments

Policies The employed hybrid planners combine a static policy (π_S) and a dynamic policy (π_D) from [17] to equip the target system with self-adaptation capabilities. The policies conform to the definition of the static and dynamic adaptation policies in Section II-A respectively and use the utility function U_{znn} as their objective function. π_S uses design-time estimations for EU–see Section III. Therefore, at each state s, for each applicable a, the expected effect on EU is predetermined. We have shown in [16] that this policy is sub-optimal in terms of overall utility but fast in terms of adaptation time. π_D uses the IBM ILOG CPLEX constraint solver for selecting actions that optimize the utility. We have shown in [16] that while this policy finds the optimal target state at each adaptation step, it can exhibit long planning times. Runtime conditions trigger a switch between π_S and π_D. For example, during surges of client traffic in `Znn.com` where more adaptation issues such as `latency` occur, employing π_S that provides timely rather than optimal adaptation plans is beneficial. However, once the traffic surge calms down, the utilization of the servers can be optimized via switching to π_D. The switch from policy π_D to π_S is charged with a cost $c_{DS} = 2$ and $c_{SD} = 200$ applies to the switch in the opposite direction. The considered costs are independent of the runtime conditions and are defined relatively and based on the measurements of the policy deployment times.

TABLE II
NAU over 24 Hours

Trace	HZ$_i$	HZ$_e$	CHP$_{dtr}$	π_D	π_S
TR1	0.79	1	0.81	0.51	0.33
TR2	0.55	0.83	1	0.49	0.38
TR3	0.76	1	0.85	0.55	0.42

Setting The experiments are repeated and averaged over 1000 simulation runs. The reported utility values are Normalized Accumulated Utility (NAU) with $NAU = U_{znn} - c_{ij}$. We executed CHP$_{dtr}$ with $RT_{thr} = 1$ sec. The same is set as the initial value to define the *high* range for $RT_{\mathcal{I}}$ in the HypeZon variants. $RT_{min} = 0.1$ sec is used as the initial value for the *optimal range* and the $RT_{\mathcal{I}}$ values in between are considered as *inRange*–see Algorithm 1. Note that the threshold values in the HypeZon variants are not deterministic and may change at runtime. The size of the look-ahead, planning, and execution horizons in CHP$_{dtr}$ is set to ∞, thus, CHP$_{dtr}$ plans for *all* the existing issues and executes the *complete* plan.

`Znn.com` is equipped with self-adaptive properties via adding a MAPE-K loop, henceforth *adaptation loop*, to the system. HZ$_e$ is realized in an additional loop that is added on top of the adaptation loop–see Fig. 4-(a). HZ$_i$ implements the internal design for meta-self-awareness–see Fig. 4-(b). The sampling interval $\mathcal{I}$ for HZ$_e$ indicates that the meta-awareness loop in the HZ$_e$ is executed once for every $\mathcal{I}$ executions of the adaptation loop. In HZ$_i$ however, $\mathcal{I} = 1$ as the hybrid planner is embedded in the adaptation loop and is executed at the same frequency as the adaptation loop–see Section V-A.

Results Table II shows NAU during 24 hours of the input traces for `Znn.com`. π_D and π_S exhibit basic planning for SAS and do not employ any hybrid planning, therefore, $c_{ij} = 0$. HZ$_e$ is executed with $\mathcal{I} = 5$. Table III presents results of sensitivity analysis for NAU obtained by HZ$_e$ with

TABLE III
NAU OF HZ_e WITH DIFFERENT $\mathcal{I}$ OVER 24 HOURS

	$\mathcal{I}=1$	$\mathcal{I}=2$	$\mathcal{I}=3$	$\mathcal{I}=4$	$\mathcal{I}=5$	$\mathcal{I}=10$	$\mathcal{I}=15$
Znn.com - TR1	0.79	0.81	0.85	0.92	1	0.43	0.25

different execution intervals $\mathcal{I}$. Note that HZ_e with $\mathcal{I}=1$ is identical to HZ_i.

RQ1 how do internal and external designs for meta-self-awareness affect HYPEZON? As shown in Table II, in two out of the three experiments, HZ_e achieves higher accumulated utility compared to HZ_i. The reason is that HZ_e has relatively larger sampling intervals, i.e., execution timescale, that provides an extended monitoring period. HZ_i holds a localized view of the system load that is limited to its relatively small monitoring period, i.e., since the last execution of the adaptation loop, and sets its control parameters and switches the employed policies accordingly. The results in Table II suggest that the relatively small execution timescale of the meta-awareness loop in HZ_i may lead to pre-mature decisions due to insufficient and localized information and, as a result, the hybrid planner is likely to demonstrate nervous and volatile behavior regarding policy switch. The results in Table II and III suggest that the utility of the hybrid planners is affected by the characteristics of the input traces. Moreover, as confirmed by Table III, larger values for execution intervals of HZ_e result in sub-optimal adaptations; HZ_e with $\mathcal{I}=15$ achieves only 25% of the optimal utility for TR1.

RQ2 how does HYPEZON perform in comparison to a deterministic hybrid planner? Table II shows that for TR2 in Znn.com, CHP_{dtr} obtains higher accumulated utility compared to the HYPEZON variants. Analysis of the TR2 characteristics revealed that the request arrival rate in TR2 is only 20% of the one in TR1 and 36% of TR3. The results together with the analysis of the input trace characteristics confirm the fact that runtime conditions, i.e., characteristics of the input traces significantly affect the utility of the adaptation. The results of Table II show that in two out of the three experiments, HZ_e outperformed HZ_i and CHP_{dtr}. For the trace with less extreme characteristics, i.e., TR2, the predefined values of the control parameters and thresholds in CHP_{dtr} are beneficial and outperform the HYPEZON variants. Overall, results suggest that HZ_e suits best the volatile operation conditions, e.g., input traces with more extreme characteristics.

Fig. 5 presents average request response times for clients in Znn.com over 60 min. HZ_e is executed with $\mathcal{I}=5$. Each measurement at sampling interval j shows the average response times (bullets) as well as the maximum and minimum response times (vertical bars) during the last 5 min. Note

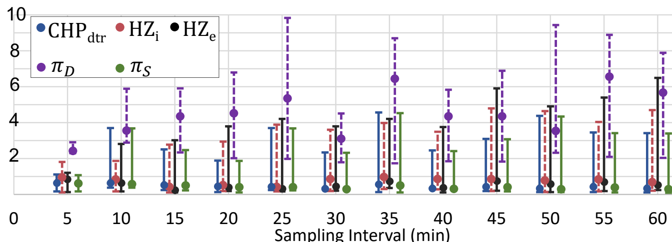

Fig. 5. Average-max-min request response time (sec) in Znn.com.

TABLE IV
NAU OVER 60 MIN FOR ZNN.COM

HZ_i	HZ_e	CHP_{dtr}	π_D	π_S
0.79	1	0.65	0.59	0.51

that the hybrid planning overhead also affects the response time values. Table IV shows the corresponding NAU of the planners at the end of 60 min. CHP_{dtr} uses predefined and deterministic values for the control parameters, thus, compared to the HYPEZON variants, has a smaller planning overhead and exhibits smaller response times. In addition to the three hybrid planners, Fig. 5 also includes response times for their constituting individual adaptation policies, i.e., π_D and π_S. π_S has similar response times to CHP_{dtr}. This is due to the deterministic decision-making in CHP_{dtr} that as soon as response time raises above 1 sec, CHP_{dtr} switches to the static policy for planning. Compared to CHP_{dtr}, HZ_e has slightly higher average response times. However, as shown in Table IV, HZ_e obtains 35% higher accumulated utility over 60 min in the same experiment. Response time values for HZ_i are higher than HZ_e and CHP_{dtr}. Despite its higher response times, Table IV shows that HZ_i obtains 14% higher accumulated utility compared to CHP_{dtr}.

RQ3 what are the effects of hybrid planning on the quality and timeliness of the adaptation? The NAU values in Table II suggest that employing hybrid planning improves the utility of a SAS compared to their individual constituting policies. π_D in Fig. 5 exhibits the highest response time, it also has significantly high *maximum* response time values, i.e., 10 sec. In case of other planners, the maximum response time does not exceed 5 sec. While results in Fig. 5 show relatively low response times for π_S, it obtains only 51% of the optimal accumulated utility in Table IV .

The baseline solutions that employ a single planner are likely to exhibit sub-optimal behavior caused by the changing operation condition. We have shown in [15] that the choice of the adaptation policy in a SAS should be steered with respect to the characteristics of the input trace, otherwise, the employed policy may render sub-optimal at runtime. Thus, as also confirmed by our experiments, employing a hybrid planner, utilizing either deterministic or adjustable parameters, results in improvements of the utility (Table II and IV) as well as the timeliness (Fig. 5) of the adaptation.

C. Discussion

There exist various ways to provide the hybrid planner in a SAS with history of the system past executions and observations to enable more informed decisions, i.e., history-aware self-adaptation schemes, e.g., [28]. Considering hybrid planning as a case for meta-awareness has the following advantages and limitation; the external design provides for a global view on the target system and the adaptation process. This allows for observing phenomena with global scope that are not observable at the awareness level. HZ_e supports explicit separation of concerns at the architecture level and allows for re-usability, easier maintenance, and independent evolution of each level. Consequently, separate

and independent mechanisms for observing, analyzing, and reasoning logic may be employed by each level. The internal design in HZ_i limits the controller in the meta-awareness subject to a localized view of its object. In this design, the meta-awareness logic is dispersed throughout the awareness level. The intertwined realization of the awareness and meta-awareness together with the embedded and dispersed meta-awareness logic makes it challenging to reason about the outcome of the meta-awareness, making composability and re-usability difficult to achieve.

VII. Conclusion and Future Work

In this paper, we presented HypeZon, a solution for hybrid planning in SAS. HypeZon leverages receding horizon control to utilize runtime information for decision-making. We proposed two alternative designs conforming to the meta-self-aware architectures to engineer HypeZon for SAS. We showed that hybrid planning, realized either as meta-awareness capabilities, or as a basic deterministic heuristic, is beneficial for SAS as it provides extended control flexibility at runtime. Our experiments suggest that, compared to the deterministic alternative, hybrid planners that utilize runtime information to dynamically adjust their decision-making are more beneficial in copping with the volatile operation conditions. Moreover, HypeZon has the characteristics of a generic hybrid planner and considers the adaptation policies as black-box and can coordinate arbitrary adaptation policies. Investigating the concurrent execution of adaptation policies in HypeZon is a subject of future work. Moreover, in addition to the functional aspects, we plan to study the architectural aspects of realizing hybrid planning as a meta-self-awareness property in SAS.

References

[1] K. Angelopoulos, A. V. Papadopoulos, V. E. Silva Souza, and J. Mylopoulos. Model predictive control for software systems with cobra. In *SEAMS*, pages 35–46, 2016.

[2] K. Angelopoulos, A. V. Papadopoulos, V. E. S. Souza, and J. Mylopoulos. Engineering self-adaptive software systems: From requirements to model predictive control. *TAAS*, 13(1):1–27, 2018.

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

[4] A. Bauer, N. Herbst, S. Spinner, A. Ali-Eldin, and S. Kounev. Chameleon: A hybrid, proactive auto-scaling mechanism on a level-playing field. *TPDS*, 30(4):800–813, 2018.

[5] R. D. Caldas, A. Rodrigues, E. B. Gil, G. N. Rodrigues, T. Vogel, and P. Pelliccione. A hybrid approach combining control theory and AI for engineering self-adaptive systems. In *SEAMS*, pages 9–19, 2020.

[6] T. Chen and R. Bahsoon. Self-adaptive and online QoS modeling for cloud-based software services. *TSE*, 43(5):453–475, 2017.

[7] T. Chen, F. Faniyi, R. Bahsoon, P. Lewis, X. Yao, L. L. Minku, and L. Esterle. The Handbook of Engineering Self-Aware and Self-Expressive Systems. *CoRR*, abs/1409.1793, May 2015. version v3.

[8] S.-W. Cheng and D. Garlan. Stitch: A Language for Architecture-Based Self-Adaptation. *Journal of Systems and Software, Special Issue on State of the Art in Self-Adaptive Systems*, 85(12), 2012.

[9] S.-W. Cheng, D. Garlan, and B. Schmerl. Making self-adaptation an engineering reality. In *Self-star Workshop*, pages 158–173. Springer, 2004.

[10] R. De Lemos, D. Garlan, C. Ghezzi, H. Giese, J. Andersson, M. Litoiu, B. Schmerl, D. Weyns, L. Baresi, N. Bencomo, et al. Software engineering for self-adaptive systems: Research challenges in the provision of assurances. In *Software Engineering for Self-Adaptive Systems III. Assurances*, pages 3–30. Springer, 2017.

[11] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[12] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio. Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements. In *ASE*, pages 283–292. IEEE, 2011.

[13] A. Filieri, M. Maggio, K. Angelopoulos, N. D'Ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein, F. Krikava, S. Misailovic, A. V. Papadopoulos, S. Ray, A. M. Sharifloo, S. Shevtsov, M. Ujma, and T. Vogel. Software engineering meets control theory. In *SEAMS*, pages 71–82, 2015.

[14] S. Ghahremani. World cup 98 website access logs. http://doi.org/10.5281/zenodo.5145855, 2021. (accessed 2021-07-29).

[15] S. Ghahremani and H. Giese. Evaluation of self-healing systems: An analysis of the state-of-the-art and required improvements. *Computers*, 9(1), 2020.

[16] S. Ghahremani, H. Giese, and T. Vogel. Efficient utility-driven self-healing employing adaptation rules for large dynamic architectures. In *2017 ICAC*, pages 59–68, 2017.

[17] S. Ghahremani, H. Giese, and T. Vogel. Improving scalability and reward of utility-driven self-healing for large dynamic architectures. *ACM Trans. Auton. Adapt. Syst.*, 14(3), Feb. 2020.

[18] H. Ghanbari, M. Litoiu, P. Pawluk, and C. Barna. Replica placement in cloud through simple stochastic model predictive control. In *CLOUD*, pages 80–87. IEEE, 2014.

[19] H. Giese, T. Vogel, A. Diaconescu, S. Götz, and S. Kounev. *Architectural Concepts for Self-Aware Computing Systems*, chapter 5, pages 109–147. Springer International Publishing, Berlin Heidelberg, Germany, 2017.

[20] C.-E. Hrabia, P. M. Lehmann, and S. Albayrak. Increasing self-adaptation in a hybrid decision-making and planning system with reinforcement learning. In *COMPSAC*, volume 1, pages 469–478. IEEE, 2019.

[21] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *FGCS*, 27(6):871–879, 2011.

[22] I. D. Landau, R. Lozano, M. M'Saad, and A. Karimi. *Adaptive control: algorithms, analysis and applications*. Springer Science & Business Media, 2011.

[23] P. Lewis, K. Bellman, C. Landauer, J. Camara, H. Giese, N. Bencomo, A. Diaconescu, and L. Esterle. *Towards a Framework for the Levels and Aspects of Self-Aware Computing Systems*, pages 51–85. Springer Verlag, Berlin Heidelberg, Germany, 2017.

[24] A. Pandey, G. A. Moreno, J. Cámara, and D. Garlan. Hybrid planning for decision making in self-adaptive systems. In *SASO*, pages 130–139. IEEE, 2016.

[25] A. Pandey, I. Ruchkin, B. Schmerl, and J. Cámara. Towards a formal framework for hybrid planning in self-adaptation. In *SEAMS*, pages 109–115. IEEE, 2017.

[26] T. Patikirikorala, A. Colman, J. Han, and L. Wang. A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *SEAMS*, pages 33–42. IEEE, 2012.

[27] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[28] L. Sakizloglou, S. Ghahremani, T. Brand, M. Barkowsky, and H. Giese. Towards highly scalable runtime models with history. In *SEAMS*, SEAMS '20, page 188–194, 2020.

[29] D. E. Seborg, D. A. Mellichamp, T. F. Edgar, and F. J. Doyle III. *Process dynamics and control*. John Wiley & Sons, 2010.

[30] A. M. Sharifloo, A. Metzger, C. Quinton, L. Baresi, and K. Pohl. Learning and evolution in dynamic software product lines. In *SEAMS*, pages 158–164. IEEE, 2016.

[31] F. Trollmann, J. Fähndrich, and S. Albayrak. Hybrid adaptation policies: towards a framework for classification and modelling of different combinations of adaptation policies. In *SEAMS*, pages 76–86, 2018.

[32] T. Vogel and H. Giese. Adaptation and abstract runtime models. In *ICSE Workshop on SEAMS*, pages 39–48, 2010.

[33] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility functions in autonomic systems. In *ICAC 2004*, pages 70–77, May 2004.