Formal Composition of Robotic Systems as Contract Programs

Mason Nakamura[1], Justin Svegliato[2], Samer B. Nashed[3], Shlomo Zilberstein[3], and Stuart Russell[2]

***Abstract*— Robotic systems are often composed of modular algorithms that each perform a specific function within a larger architecture, ranging from state estimation and task planning to trajectory optimization and object recognition. Existing work for specifying these systems as a formal composition of contract algorithms has limited expressiveness compared to the variety of sophisticated architectures that are commonly used in practice. Therefore, in this paper, we (1) propose a novel metareasoning framework for formally composing robotic systems as a contract program with programming constructs for functional, conditional, and looping semantics and (2) introduce a recursive hill climbing algorithm that finds a locally optimal time allocation of a contract program. In our experiments, we demonstrate that our approach outperforms baseline techniques in a simulated pick-and-place robot domain.**

I. Introduction

Robotic systems are often composed of modular algorithms that each perform a specific function within a larger architecture, ranging from state estimation [28] and task planning [9], [6] to trajectory optimization [13] and object recognition [14]. However, in practice, there is often no formal model that represents the relationship between how the results of these algorithms can augment or diminish each other's performance. For example, a pick-and-place robot may have an upstream algorithm for state estimation and a downstream algorithm for task planning but no formal model that represents the relationship between the results of these algorithms. Therefore, since robotic systems have been increasingly deployed across a range of complex real-world domains, there has been a growing need to formally model their architectures as a network of algorithms in order to improve their utility and performance during operation.

A promising approach to formally modeling the architecture of a robotic system is to represent it as a *composition* of *contract algorithms* called a *contract program* [31], [35]. In short, a contract algorithm is an algorithm that (1) gradually improves the quality of its current solution at runtime but (2) must be terminated at a fixed deadline [32]. Consequently, in this approach, once a robotic system has been specified as a composition of contract algorithms, the objective is to determine a time allocation for each contract algorithm such that the overall expected utility of the robotic system is optimized subject to a given computation time budget.

This work was supported by NSF grants IIS-1813490/IIS-1954782 and a gift from the Open Philanthropy Foundation.

[1]Marist College, Poughkeepsie, NY, USA. Emails: mason.nakamura1@marist.edu

[2]University of California, Berkeley, CA, USA. Emails: {jsvegliato, russell}@berkeley.edu

[3]University of Massachusetts, Amherst, MA, USA. Emails: {snashed, shlomo}@cs.umass.edu

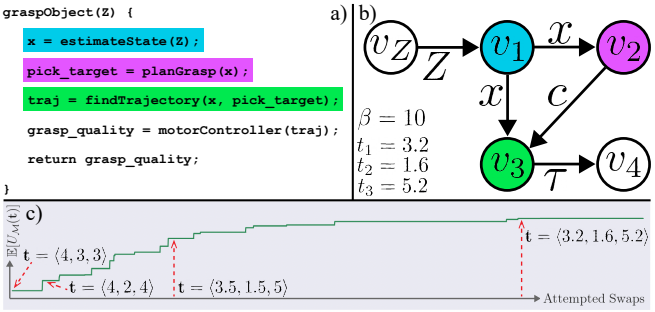

Fig. 1. Our approach for an object grasping contract program $\mathcal{M}$ within the architecture of a robotic system. (a) shows the original `graspObject` function specified in a high-level program written in a language like C++ or Python that invokes three contract algorithms `estimateState`, `planGrasp`, and `findTrajectory` along with the execution algorithm `motorController`. (b) shows the corresponding contract program with three vertices v_1, v_2, and v_3 for the contract algorithms that each can be allocated a computation time t_i where the sum $3.2 + 1.6 + 5.2$ equals a time budget $\beta = 10$. (c) shows the recursive hill climbing algorithm that gradually improves a time allocation $\mathbf{t} = \langle 4, 3, 3 \rangle$ in term of its expected utility $\mathbb{E}[U_{\mathcal{M}}(\mathbf{t})]$ with each attempted swap until converging to a locally optimal time allocation $\vec{t}^{*} = \langle 3.2, 1.6, 5.2 \rangle$ given a utility function U.

However, while this approach offers a framework for formally modeling the architecture of a robotic system as a composition of contract algorithms, it is has limited expressiveness compared to the variety of sophisticated architectures that are commonly used in practice. In particular, this approach only considers a contract program with *functional* semantics: the output of a set of upstream contract algorithms can only be used as input to a downstream contract algorithm. Ideally, in order to formally model complex robotic systems and improve their utility and performance in complex real-world domains, it is critical to build contract programs with programming constructs for not only *functional* semantics but also *conditional* and *looping* semantics.

Therefore, we propose a novel metareasoning framework illustrated in Figure 1 for formally modeling a robotic system as a contract program with programming constructs for not only functional but also conditional and looping semantics. By building a performance profile for each programming construct representing the probability of a solution quality being generated by the contract algorithm for a given time allocation and a set of input qualities, it is possible to specify a formal representation of the expected utility for a contract program. Given this representation, a contract program can be solved—a time allocation can be determined—by using a recursive hill climbing algorithm that finds a locally optimal time allocation. Empirically, we show that our metareasoning approach is effective at determining the time allocations of a contract program, outperforming simple baseline techniques, in a simulated pick-and-place robot domain.

II. Related Work

There has been a large body of research on metareasoning. The most relevant work to this paper is on anytime algorithms, a class of algorithms encapsulating *interruptible* and *contract* algorithms [32]. The main property of an anytime algorithm is that it gradually improves the quality of its current solution at runtime. To obtain the current solution, an *interruptible* algorithm can be interrupted at any time while a *contract* algorithm must be terminated at a fixed deadline. Generally, it is has been shown that any *interruptible* algorithm can be a contract algorithm by executing it until fixed deadline while any *contract* algorithm can be an interruptible algorithm by repeatedly executing it with a fixed deadline that grows exponentially [35]. In robotics specifically, anytime algorithms have been shown to be effective in a variety of domains, ranging from state estimation [28] and task planning [9], [6] to trajectory optimization [13] and object recognition [14]. We outline a few relevant areas of metareasoning that have been developed for anytime algorithms and related domains below.

First, there has been work on metareasoning for optimal stopping of interruptible algorithms. The earliest approach, namely *fixed allocation*, executes the anytime algorithm until a stopping point determined prior to runtime [11], [5]. Although fixed allocation is effective when there is negligible uncertainty in the performance of the anytime algorithm, there is often substantial uncertainty in real-time planning problems [19]. Hence, a more sophisticated approach, namely *monitoring and control*, tracks the performance of the anytime algorithm and estimates a stopping point at runtime [12], [34], [10], [16], [26], [23], [21].

Next, there has been work on metareasoning for optimal parameter tuning of anytime algorithms that has been designed for specific anytime algorithms. For example, for task planning, methods tune the weight of an anytime heuristic search algorithm called *anytime weighted A** by selecting the best weight for a problem [9], choosing the best weight for an instance of a problem [20], modifying the weight heuristically [27], or adjusting the weight randomly [3]. Moreover, for motion planning, methods tune the growth factor of an anytime motion planning algorithm called *RRT** [29], [1], [15]. Most recently, it has been shown that reinforcement learning is an effective framework for learning how to tune the parameters of any anytime algorithm [4], [2].

Finally, there has been work on metareasoning for optimal selection among a portfolio of decision-making models or algorithms. This work recognizes that different models or algorithms tend to dominate each other on different types of problems. For example, methods based on metareasoning have been developed to select the best model needed to recover from a range of exceptions or safety concerns [25], [24], [22], the best algorithm needed to solve challenging computational problems like constraint satisfaction problems [33], [8], [30], the best abstraction needed to solve Markov decision processes [18], [17], and the best node expansion to be performed in heuristic search algorithms [7].

III. Formal Composition of Robotic Systems

We begin by introducing a novel metareasoning framework for formally composing a robotic system as a contract program. The rest of this section proceeds as follows. First, we define the formal metareasoning problem for contract programs. Next, we offer foundational programming constructs for the control flow of a contract program. Finally, we propose a recursive hill climbing algorithm that finds a locally optimal time allocation of a contract program.

*1) **Metareasoning for contract programs**:* First, we define the formal metareasoning problem for contract programs. A *contract program* is a composition of contract expressions that interact with each other through a set of well-defined programming constructs. Like any high-level program, the control flow of a contract program can use programming constructs like functional, conditional, and looping expressions. Formally, a contract program can be represented by a directed graph where each vertex denotes a contract expression and each edge denotes an interaction between contract expressions. We define a contract program below and then define each expression later in the paper.

Definition 1. *A **contract program** $\mathcal{M}$ is a composition of contract expressions represented by a directed graph:*

- *A vertex v_Z is an **input variable** Z.*
- *A vertex v_F is a **functional expression** $F(G_1, \ldots, G_n)$ such that the vertices $v_{G_1}, \ldots, v_{G_n}$ with edges directed into vertex v_F denote the set of input variables or functional expressions $G_1, \ldots, G_n$.*
- *A vertex v_C is a **conditional expression** $C(F_C, F_{\neg C})$ with a condition C such that the vertices v_{F_C} and $v_{F_{\neg C}}$ with edges directed into vertex v_C denote the functional expressions F_C and $F_{\neg C}$ such that C is* TRUE *with probability ρ and* FALSE *with probability $1 - \rho$.*
- *A vertex v_L is a **looping expression** $L(F)$ such that the vertex v_F with an edge directed into vertex v_L denotes σ sequential invocations of the functional expression F such that σ is selected offline or dynamically online.*

The notation $v \in \mathcal{V}[\mathcal{M}]$ denotes a vertex, $e \in \mathcal{E}[\mathcal{M}]$ denotes an edge, and $\alpha \in \mathcal{A}[\mathcal{M}]$ denotes a contract algorithm.

It is possible to calculate the expected utility of a contract program for a given time allocation. This is computed by summing over the probability of the final solution qualities generated by each contract expression for a given time allocation multiplied by the utility that results from the final solution qualities. We define this expected utility below where T is a normed vector space representing the set of possible time allocations and $Q_i = [0, 1]$ is a continuous set of final solution qualities for each contract algorithm α_i.

Definition 2. *The **expected utility** $\mathbb{E}[U_\mathcal{M}] : T \longrightarrow \mathbb{R}$ of a contract program $\mathcal{M}$ for a given time allocation $\mathbf{t} \in T$ is:*

$$\mathbb{E}[U_\mathcal{M}(\mathbf{t})] = \sum_{(q_1,\ldots,q_n)\in Q_1\times\ldots\times Q_n} \Pr(q_1, \ldots, q_n \mid \mathbf{t}) U(q_1, \ldots, q_n),$$

where $(q_1, \ldots, q_n) \in Q_1 \times \ldots \times Q_n$ are the final solution qualities generated by each contract expression and $U : Q_1 \times$

$\cdots \times Q_n \to \mathbb{R}$ *is the utility function of the problem being solved by the contract program.*

The objective of the metareasoning problem for contract programs is to determine the optimal time allocation of the contract program that optimizes the expected utility subject to a given time budget. We define this objective below.

Definition 3. *A time allocation* $\mathbf{t}^* = \langle t_1, t_2, \ldots, t_{|\mathcal{V}[\mathcal{M}]|} \rangle$ *of a contract program* $\mathcal{M}$ *is* ***optimal*** *if and only if:*

$$\mathbf{t}^* = \arg\max_{\mathbf{t} \in T} \mathbb{E}\left[U_{\mathcal{M}}(\mathbf{t})\right],$$

where $T = \{\mathbf{t} \in (\mathbb{R}^+)^{|\mathcal{V}[\mathcal{M}]|} \mid \|\mathbf{t}\|_1 = \beta\}$ *is the space of time allocations that are available to be allocated to each contract algorithm of the contract program given a time budget* β*.*

2) ***Extending contract programs****:* Next, we offer foundational programming constructs for the control flow of a contract program. For each programming construct, we give its semantics and summarize its performance with a *performance profile* that represents the probability of the final solution quality generated by the contract algorithm for a given time allocation and a set of input solution qualities. This is typically provided by a domain expert or built through simulations of the contract algorithm offline [32].

Functional expressions, like function statements in high-level programs that were introduced in early work on contract programs [35], invoke a contract algorithm on a set of inputs:

Definition 4. *A* ***functional expression*** F *is a function invocation* $F(G_1, \ldots, G_n)$ *that invokes a contract algorithm* $\alpha \in \mathcal{A}[\mathcal{M}]$ *on a set of input variables or functional expressions* $G_1, \ldots, G_n$*.*

Definition 5. *A* ***functional expression performance profile*** $\lambda : \mathbf{Q} \times T \longrightarrow \Delta^{|Q|}$ *gives the probability* $\lambda(q \mid \mathbf{q}, t)$ *of a functional expression* F *generating an output quality* $q \in Q$ *given a time allocation* $t \in T$ *and a tuple of input qualities* $\mathbf{q} \in \mathbf{Q} = Q_1 \times \cdots \times Q_n$*.*

Conditional expressions, analogous to if-else statements in high-level programs, invoke either a left or right functional expression based on whether the condition is true or false:

Definition 6. *A* ***conditional expression*** $C(F_C, F_{\neg C})$ *with a condition* C *that evaluates in* τ *computation time to* TRUE *with probability* ρ *or* FALSE *with probability* $1 - \rho$ *is:*

$$\mathbb{1}[C = \textsc{True}] \cdot F_C(\cdot) + \mathbb{1}[C = \textsc{False}] \cdot F_{\neg C}(\cdot),$$

where $F_C(\cdot)$ *and* $F_{\neg C}(\cdot)$ *are functional expressions.*

Definition 7. *A* ***conditional expression performance profile*** $\lambda : Q_{F_C} \times Q_{F_{\neg C}} \times \mathbf{Q} \times T \longrightarrow \Delta^{|Q|}$ *gives the probability*

$$\lambda(q \mid q_{F_C}, q_{F_{\neg C}}, \mathbf{q}, t) = \rho \cdot \lambda_{F_C}(q_{F_C} \mid \mathbf{q}, t - \tau) + (1 - \rho) \cdot \lambda_{F_{\neg C}}(q_{F_{\neg C}} \mid \mathbf{q}, t - \tau)$$

of a conditional expression C *generating an output quality* $q \in Q$ *given a time allocation* $t \in T$*, a tuple of input qualities* $\mathbf{q} \in \mathbf{Q} = Q_1 \times \cdots \times Q_n$*, and a pair of left and right branch qualities* $q_{F_C} \in Q_{F_C}$ *and* $q_{F_{\neg C}} \in Q_{F_{\neg C}}$ *for functional expressions* $F_C(\cdot)$ *and* $F_{\neg C}(\cdot)$*.*

Algorithm 1: RECURSIVEHILLCLIMBING$(\cdot)$

In: A contract program $\mathcal{M}$, a time budget $\beta \in \mathbb{R}^+$, and an initial time allocation vector $\mathbf{t}^0 \in (\mathbb{R}^+)^{|\mathcal{V}[\mathcal{M}]|}$
Out: A final time allocation vector $\mathbf{t} \in (\mathbb{R}^+)^{|\mathcal{V}[\mathcal{M}]|}$
Require: A performance profile λ_α for each contract algorithm $\alpha \in \mathcal{A}[\mathcal{M}]$, a resolution threshold $\eta \in \mathbb{R}^+$, and a refinement rate $\psi \in (1, \infty)$

```
ε ← ‖t^0‖_∞
t ← t^0, t' ← t^0
while ε ≥ η do
    for (v_i, v_j) in V[M]^2 such that v_i ≠ v_j do
        α_i ← CALCULATETIMEALLOCATION(v_i)
        α_j ← CALCULATETIMEALLOCATION(v_j)
        if α_i − ε < 0 or α_j + ε > β then
            continue
        β_i ← α_i − ε, β_j ← α_j + ε
        for v_k in {v_i, v_j} do
            if v_k.ISFUNCTIONALEXPRESSION() then
                t'_k ← β_k
            if v_k.ISCONDITIONALEXPRESSION() then
                RHC(M_C, β_k, (β_k/α_k) t_{v_C})
                RHC(M_¬C, β_k, (β_k/α_k) t_{v_¬C})
            if v_k.ISLOOPINGEXPRESSION() then
                RHC(M_L, β_k, (β_k/α_k) t_{v_L})
        if E[U_M(t')] > E[U_M(t)] then
            t ← t'
    if t = t' then
        ε ← ε/ψ
return t
```

Looping expressions, similar to for and while statements in high-level programs, invoke a functional expression for a number of sequential invocations that can be specified offline or dynamically online:

Definition 8. *A* ***looping expression*** $L(F)$ *that iterates for a number of sequential invocations* $\sigma > 0$ *that can be specified by hand offline or dynamically online is:*

$$F^{(\sigma)} \circ F^{(\sigma-1)} \circ \ldots \circ F^{(1)}(I),$$

where $F^{(i)}$ *is the* i*th intermediate sequential invocation of the functional expression* F *and* I *is the set of initial input variables and functional expressions.*

Definition 9. *A* ***looping expression performance profile*** $\lambda : \mathbf{Q} \times \mathbf{T} \longrightarrow \Delta^{|Q|}$ *gives the probability*

$$\lambda(q \mid \mathbf{q}, \mathbf{t}) = \prod_{i=1}^{\sigma} \lambda_{F^{(i)}}(q^{(i)} \mid q^{(i-1)}, \mathbf{q}, t^{(i)})$$

of a looping expression L *generating an output quality* $q \in Q$ *given a tuple of intermediate time allocations* $\mathbf{t} \in \mathbf{T} = T_1 \times \cdots \times T_\sigma$*, a tuple of initial input qualities* $\mathbf{q} \in \mathbf{Q} = Q_1 \times \cdots \times Q_n$*, and an output quality* $q^{(i)}$ *for the* i*th intermediate invocation of functional expression* F*.*

Importantly, we note that any method for computing the expected utility $\mathbb{E}[U_{\mathcal{M}}(\mathbf{t})]$ of a contract program $\mathcal{M}$ for a given time allocation $\mathbf{t} \in T$ depends on the joint probability $\Pr(q_1, \ldots, q_n \mid \mathbf{t})$ of the solution qualities $(q_1, \ldots, q_n) \in Q_1 \times \ldots \times Q_n$. By expressing this joint probability as a chain of conditional probabilities (similar to a Bayesian network),

this can be computed efficiently by using the performance profile λ_α of each contract algorithm $\alpha \in \mathcal{A}[\mathcal{M}]$.

*3) **Solving contract programs**:* Finally, we propose a recursive hill climbing algorithm that finds a locally optimal time allocation of a contract program. Algorithm 1 describes the RECURSIVEHILLCLIMBING algorithm that takes in a contract program, a time budget, and an initial time allocation vector and returns a final time allocation vector. At a high level, this method is a local search algorithm that exchanges a small unit of time between different pairs of vertices until the expected utility of the contract program converges.

We describe Algorithm 1 in detail below. The resolution, current time allocation vector, and proposed time allocation vector are initialized (Lines 1-2). While the resolution is greater than or equal to the resolution threshold, a loop iterates over each two-permutation of vertices (Lines 3-4). The time allocation of each vertex is extracted from its contract program (Lines 5-6). If the time allocation exchange is valid, each time allocation is calculated by either adding or subtracting the resolution (Lines 7-9). For each vertex, there are two cases. If it is a functional expression, its time allocation is updated. If it is a conditional or looping expression, the algorithm is invoked recursively (Lines 10-17). Here, we see that the functional expression are the base case because conditional and looping expressions are composed of functional expressions. After that, if the proposed time allocation vector has an expected utility higher than the current time allocation vector, the current vector is replaced with the proposed vector (Lines 18-19). Finally, if there is no change, the resolution is contracted by the refinement rate to allow for fine-grained adjustments (Lines 20-21). A final time allocation vector is returned (Line 22).

We provide the worst-case time and space complexity of the RECURSIVEHILLCLIMBING algorithm below.

Proposition 1 (Time and Space Complexity)**.** *Algorithm 1 has a worst-case time complexity of* $O(2\log_\psi(\frac{\epsilon}{\eta}) \cdot |\mathcal{V}[\mathcal{M}]|^3)$ *and a worst-case space complexity of* $O(|\mathcal{V}[\mathcal{M}]|)$.

Proof Sketch. We begin by proving the worst-case time complexity of Algorithm 1. First, observe that the while loop iterates as long as the condition $\epsilon \geq \eta$ holds between the resolution ϵ and the resolution threshold η. Moreover, the resolution ϵ is contracted by the refinement rate ψ in each iteration of the while loop. Hence, as the while loop iterates x times according to the equation $\frac{\epsilon}{\psi^x} = \eta$, the while loop iterates $x = \log_\psi(\frac{\epsilon}{\eta})$ times. Next, the for loop iterates at most $|\mathcal{V}[\mathcal{M}]|^2$ times for each pair of $|\mathcal{V}[\mathcal{M}]|$ vertices. Finally, each iteration of the for loop results in at most $2|\mathcal{V}[\mathcal{M}]|$ recursive invocations of the algorithm. Therefore, when multiplying each of these terms together, the worst-case time complexity of Algorithm 1 is $O(2\log_\psi(\frac{\epsilon}{\eta}) \cdot |\mathcal{V}[\mathcal{M}]|^3)$.

We now prove the worst-case space complexity of Algorithm 1. Observe that only two time allocation vectors $\mathbf{t}$ and $\mathbf{t}'$, each of size $|\mathcal{V}[\mathcal{M}]|$, are stored. Therefore, using an optimized *in-place* memory implementation, the worst-case space complexity of Algorithm 1 is $O(|\mathcal{V}[\mathcal{M}]|)$. □

TABLE I

THE CONTRACT ALGORITHMS FOR THE PICK-AND-PLACE ROBOT.

Vertex	Input	Output	Function
v_{12}	Z	s_0	Estimate current task state
$v_{11}-v_9$	S^i	S^{i+1}	Expand ply of task search tree
v_8	S^k	Υ, c_1	Select parameterized grasp type
v_7	Υ, c_1	$\tau_{c_0}^{c_1}$	Move end effector to pick config
v_5	Z	x_o^2	Identify obstacles at place config
v_4	x_o^2	c_2	Select place config
v_3	x_o^2, c_1, c_2	$\tau_{c_1}^{c_2}$	Move end effector to place config
v_1	x_o^2, $\tau_{c_1}^{c_2}$, c_2	$\tau_{c_2}^{c_0}$	Move end effector to pre-grasp config
v_6	Z	x_o^1	Identify obstacles at grasp config
v_2	x_o^1, c_1	$\tau_{c_1}^{c_0}$	Move end effector to pre-grasp config

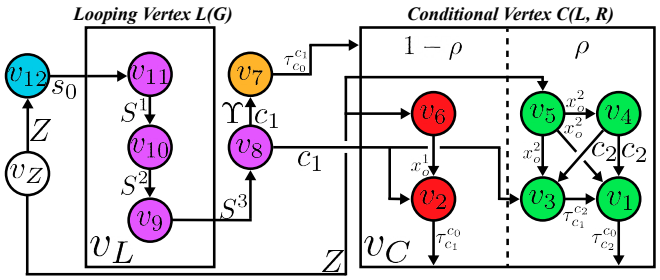

Fig. 2. A simplified directed graph representation of a contract program for the pick-and-place robot with vertices representing contract expressions and edges representing interactions between contract expressions.

IV. PICK-AND-PLACE ROBOT DOMAIN

We now turn to an example contract program for a simulated pick-and-place robot domain illustrated in Figure 2. Here, the robot has a contract program with contract algorithms that each perform a function described in Table I.

The robot begins by estimating the current state of the task (v_{12}). Next, the robot plans by repeatedly expanding a task search tree for 4 plies ($v_{11}-v_9$, v_8) in which the final ply generates a grasp type. The robot then plans/executes its grasp (v_7). Finally, depending on whether it succeeds or fails, the robot either places the object (v_5, v_4, v_3) or evaluates the workspace (v_6) prior to resetting the end effector to its initial configuration (v_1, v_2). For the input and output of each vertex, we have that: Z is the raw sensor data; s_0 is the initial state; S^i is the task search tree after i plies starting with $S^0 = \{s_0\}$ as the root task search tree; Υ is a grasp type; c_0, c_1, and c_2 are the pick/place configurations with c_0 as given offline and c_1 and c_2 computed online; τ_a^b is a trajectory from configuration a to configuration b; and x_o^i are location estimates for the target and obstacles at configuration c_i.

In a real-time intelligent system like a pick-and-place robot, each vertex is implemented with a specific contract algorithm. In particular, there are a range of algorithms that can be used as a contract algorithm for each vertex of the contract program. Monte Carlo localization [28] could be used for state estimation in vertex v_{12}. Monte Carlo tree search [6] or anytime heuristic search [9] could be used for task planning in vertices $v_{11}-v_9$ and v_8. RRT* [13] could be used for trajectory optimization in vertices v_7, v_3, v_1, and v_2. Anytime object and scene understanding [14] could be used for object recognition in vertices v_5 and v_6 and selecting a

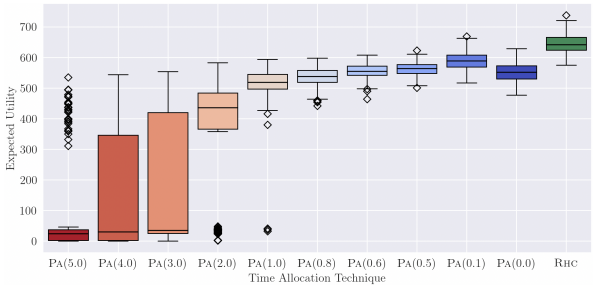

Fig. 3. The expected utilities of the RHC and PA time allocation techniques on the pick-and-place robot domain.

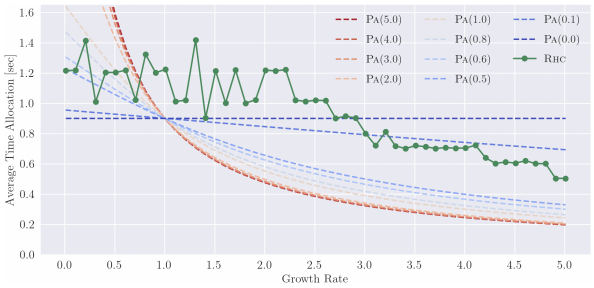

Fig. 4. The time allocations of the RHC and PA time allocation techniques on the pick-and-place robot domain for the specific contract expression v_5.

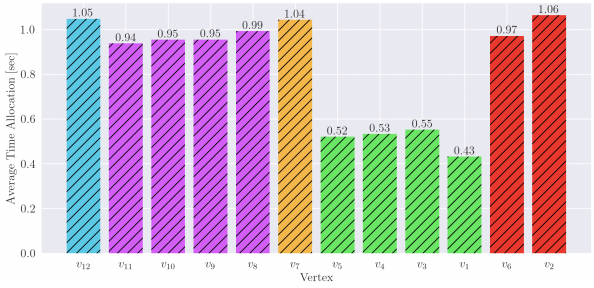

Fig. 5. The time allocations of the RHC technique across all contract expressions on the pick-and-place robot domain.

place configuration via scene recognition in v_4. Generally, these algorithms are either contract algorithms by design or can be adjusted slightly to be contract algorithms [35].

The performance profile for each contract algorithm can be built using a standard, well-studied procedure [32]: for each contract algorithm, this procedure (1) performs simulations to generate solution quality trajectories and (2) builds a performance profile from these solution quality trajectories. Given these performance profiles, the performance profiles for functional/conditional/looping expressions can be built.

Step 1: We perform ℓ_1 simulations of ℓ_2 time steps to generate solution quality trajectories for each contract algorithm α. Formally, the $i \in [1, \ell_1]$ simulations of $t \in [1, \ell_2]$ time steps each generate solution quality trajectories $\langle q_1^{(i)}, q_2^{(i)}, \ldots, q_{\ell_2}^{(i)} \rangle$ such that each solution quality $q_t^{(i)}$ is

$$q_t^{(i)} = 1 - \frac{1}{e^{(C+\xi)t}}$$

for the growth rate C of the contract algorithm α and some noise $\xi \sim \mathcal{N}(\mu, \sigma)$ drawn from a Gaussian $\mathcal{N}$ [5].

Step 2: We build the performance profile λ_α from these solution quality trajectories for each contract algorithm α. Formally, the performance profile $\lambda_\alpha(q|\mathbf{q}, t)$ is

$$\lambda_\alpha(q|\mathbf{q}, t) = \frac{\sum_{i=1}^{\ell_1} \mathbb{1}[q = q_t^{(i)}]}{\ell_1 \cdot \ell_2},$$

for a time allocation $t \in T$ and a tuple of input qualities $\mathbf{q} \in \mathbf{Q} = Q_1 \times \cdots \times Q_n$.

It is important to highlight that any "well-behaved" anytime algorithm exhibits the *diminishing returns* property [32]: the improvement in solution quality is large at earlier time steps and diminishes gradually in later time steps. Intuitively, this is due to a potential solution being easy to improve at early stages of computation but difficult to improve at later stages of computation. Naturally, the performance profile of a contract algorithm that is generated by this procedure will exhibit the diminishing returns property. As a result, while the performance profiles are simulated in the interest of simplicity in this paper, any performance profile generated from actual performance data would share a similar form with the diminishing returns property.

V. Experiments

We now demonstrate our approach (RHC) on the simulated pick-and-place robot domain and show that it is effective at determining the time allocations of a contract program.

As a baseline, we use different parameterizations of a proportional time allocation technique PA(ϕ) that determines distinct time allocations for a given contract program with a time budget of β. This technique assigns a proportional time allocation to each contract expression based on the performance profile of that contract expression. Formally, for a proportionality parameter $\phi \in \mathbb{R}_{\geq 0}$, each contract expression $v_i \in \mathcal{V}[\mathcal{M}]$ summarized by a performance profile with a growth rate C_{v_i} is assigned a computation time of

$$\left[\frac{\pi - 2 \arctan(\phi \cdot C_{v_i})}{|\mathcal{V}[\mathcal{M}]|\pi - \sum_{v_j \in \mathcal{V}[\mathcal{M}]} 2 \arctan(\phi \cdot C_{v_j})} \right] \beta.$$

Intuitively, contract expressions with a *high* growth rate are given *less* time while contract expressions with a *low* growth rate are given *more* time. Moreover, the proportionality parameter ϕ shifts how much time is assigned to each contract algorithm based on the growth rate of its performance profile.

For any contract program considered throughout our experiments, there is a time budget $\beta = 10$. Similarly, the performance profile of each contract expression is built by performing $\ell_1 = 1000$ simulations of the contract algorithm of $\ell_2 = 100$ time steps given a growth rate C following the two-step procedure described earlier in the paper.

In our experiments, we perform 150 trials in which the RHC and PA baseline techniques must determine the time allocations of the contract program. In each trial, the contract expressions of the contract program are assigned a random growth rate $C \in [0, 5]$ to simulate different performance characteristics. After each trial, the final expected utilities and time allocations of each technique are recorded.

Figure 3 provides the final expected utilities of the RHC and PA baseline techniques averaged over all trials. The RHC technique with a mean of 644.91 between a lower and upper quartile of 623.68 and 665.56 outperforms the best PA

baseline technique (0.1) with a mean of 589.28 between a lower and upper quartile of 569.19 and 607.91. Moreover, the RHC technique has a narrow range (163.01) around its mean compared to the large range across some of the PA baseline techniques, which suggests the RHC technique may be less sensitive to the variance in the performance characteristics (i.e., the growth factor) of each contract expression.

Figure 4 summarizes the final time allocations of the RHC and PA baseline techniques for the specific contract expression v_5 averaged over all trials. When the proportionality parameter ϕ is zero, the PA baseline technique assigns the same time allocation to all growth rates C. When the proportionality parameter ϕ is high (5.0), the PA baseline technique assigns decreased time allocations relative to RHC for growth rates $C > 1.0$ and increased time allocations relative to RHC for growth rates $C < 1.0$. Importantly, the RHC technique adjusts its time allocation to this contract expression based on the growth rate C: the time allocations decrease as the growth factor C increases from the minimum $C = 0.0$ to the maximum $C = 5.0$. Overall, this suggests that the RHC technique adjusts its time allocations to this contract expression based on the growth rate in order to generate a higher expected utility than the PA baseline technique.

Figure 5 describes the final time allocations of the RHC technique across all contract expressions averaged over all trials. This is to demonstrate the general trend of how the RHC technique allocates time to each contract expression based on its location within the contract program independent of their growth factors. Here, we notice that RHC determines that it is worthwhile to allocate time that roughly increases with each loop of the looping expression ($v_{11} - v_9$, v_8). Moreover, for the conditional expression, the first branch (v_5, v_4, v_3, v_1) and the second branch (v_6, v_2) must be allocated the same total time (2.03 seconds) but is done so in different ways. Finally, the individual functional expressions (v_{12}, v_7) are allocated roughly the same time (about 1.0 second).

Our approach to formally modeling robotic systems as a contract program offers many advantages. First, by reasoning about how algorithms can impact each other's performance, a robotic system does not need a time allocation specified since it can be automatically found for a given time budget. Next, by introducing standard programming constructs for conditional and looping semantics, realistic architectures can be modeled like selecting a motion planner based on the environment or performing plies of a task planner. Finally, novel programming constructs can easily be used if their performance is summarized by a performance profile. In short, this approach can formally model the architecture of a robotic system as a contract program and optimize its performance by adjusting time allocations to each contract expression and the topology of the contract program.

VI. CONCLUSION

This paper proposes a novel metareasoning framework for formally modeling the architecture of a robotic system as a contract program with programming constructs for functional, conditional, and looping semantics and then introduces a recursive hill climbing algorithm that finds a locally optimal time allocation for that contract program. Finally, we demonstrate that our approach outperforms a baseline technique in a simulated pick-and-place robot. Future work will introduce additional programming constructs from high-level programs and develop effective time allocation methods.

REFERENCES

[1] B. Akgun and M. Stilman. Sampling heuristics for optimal motion planning in high dimensions. In *IROS*, 2011.
[2] A. Bhatia, J. Svegliato, S. B. Nashed, and S. Zilberstein. Tuning the hyperparameters of anytime planning: A metareasoning approach with deep reinforcement learning. In *ICAPS*, 2022.
[3] A. Bhatia, J. Svegliato, and S. Zilberstein. On the benefits of randomly adjusting Anytime Weighted A*. In *12th SOCS*, 2021.
[4] A. Biedenkapp, H. F. Bozkurt, T. Eimer, F. Hutter, and M. Lindauer. Dynamic algorithm configuration: Foundation of a new meta-algorithmic framework. In *24th ECAI*, 2020.
[5] M. Boddy and T. L. Dean. Deliberation scheduling for problem solving in time-constrained environments. *AIJ*, 67(2), 1994.
[6] C. B. Browne, E. Powley, D. Whitehouse, et al. A survey of Monte Carlo tree search methods. *T-CIAIG*, 4(1), 2012.
[7] D. Cope, J. Svegliato, and S. Russell. Learning to plan with tree search via deep RL. In *IJCAI Workshop on Bridging the Gap Between AI Planning and Reinforcement*, 2023.
[8] C. P. Gomes and B. Selman. Algorithm portfolios. *AIJ*, 126(1-2), 2001.
[9] E. A. Hansen and R. Zhou. Anytime heuristic search. *JAIR*, 28, 2007.
[10] E. A. Hansen and S. Zilberstein. Monitoring and control of anytime algorithms: A dynamic programming approach. *AIJ*, 126(1-2), 2001.
[11] E. J. Horvitz. Reasoning about beliefs and actions under computational resource constraints. In *3rd UAI*, 1987.
[12] E. J. Horvitz. *Computation and action under bounded resources*. PhD thesis, Stanford University, CA, 1990.
[13] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli, and S. Teller. Anytime motion planning using the RRT*. In *ICRA*, 2011.
[14] S. Karayev, M. Fritz, and T. Darrell. Anytime recognition of objects and scenes. In *CVPR*, 2014.
[15] S. Kiesel, E. Burns, and W. Ruml. Abstraction-guided sampling for motion planning. In *5th SOCS*, 2012.
[16] C. H. Lin, A. Kolobov, E. Kamar, and E. Horvitz. Metareasoning for planning under uncertainty. In *24th IJCAI*, 2015.
[17] S. B. Nashed, J. Svegliato, A. Bhatia, S. Russell, and S. Zilberstein. Selecting the partial state abstractions of MDPs: A metareasoning approach with deep reinforcement learning. In *IROS*, 2022.
[18] S. B. Nashed, J. Svegliato, M. Brucato, C. Basich, R. Grupen, and S. Zilberstein. Solving Markov decision processes with partial state abstractions. In *ICRA*, 2021.
[19] C. J. Paul, A. Acharya, B. Black, and J. K. Strosnider. Reducing problem-solving variance to improve predictability. *Communications of the ACM*, 34(8), 1991.
[20] X. Sun, M. J. Druzdzel, and C. Yuan. Dynamic Weighting A* search-based MAP algorithm for Bayesian networks. In *20th IJCAI*, 2007.
[21] J. Svegliato. *Metareasoning for planning and execution in autonomous systems*. PhD thesis, University of Massachusetts Amherst, 2022.
[22] J. Svegliato, C. Basich, S. Saisubramanian, and S. Zilberstein. Metareasoning for safe decision making in autonomous systems. In *ICRA*, 2022.
[23] J. Svegliato, P. Sharma, and S. Zilberstein. A model-free approach to meta-level control of anytime algorithms. In *ICRA*, Paris, France, 2020.
[24] J. Svegliato, S. J. Witwicki, K. H. Wray, and S. Zilberstein. Introspective autonomous vehicle operational management, U.S. Patent 10,649,453, May 2020.
[25] J. Svegliato, K. H. Wray, S. J. Witwicki, J. Biswas, and S. Zilberstein. Belief space metareasoning for exception recovery. In *IROS*, 2019.
[26] J. Svegliato, K. H. Wray, and S. Zilberstein. Meta-level control of anytime algorithms with online performance prediction. In *27th IJCAI*, 2018.
[27] J. Thayer and W. Ruml. Using distance estimates in heuristic search. In *19th ICAPS*, 2009.
[28] S. Thrun, D. Fox, W. Burgard, et al. Monte Carlo localization with mixture proposal distribution. In *AAAI*, 2000.
[29] C. Urmson and R. Simmons. Approaches for heuristically biasing RRT growth. In *IROS*, 2003.
[30] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *JAIR*, 32, 2008.
[31] S. Zilberstein. *Operational rationality through compilation of anytime algorithms*. PhD thesis, University of California Berkeley, 1993.
[32] S. Zilberstein. Using anytime algorithms in intelligent systems. *AIM*, 17(3), 1996.
[33] S. Zilberstein and A.-I. Mouaddib. Optimal scheduling of progressive processing tasks. *IJAR*, 25(3), 2000.
[34] S. Zilberstein and S. J. Russell. Approximate reasoning using anytime algorithms. In *Imprecise and Approximate Computation*. Springer, 1995.
[35] S. Zilberstein and S. J. Russell. Optimal composition of real-time systems. *AIJ*, 82(1-2), 1996.