Hybrid Declarative-Imperative Representations for Hybrid Discrete-Continuous Decision-Making

Jiayuan Mao[1], Joshua B. Tenenbaum[1],
Tomás Lozano-Pérez[1], and Leslie Pack Kaelbling[1]

Massachusetts Institute of Technology, Cambridge MA 02139, USA

Abstract. We present a robot-behavior description language CDL that can express both direct imperative strategies and planning-based strategies, and combine them seamlessly within the same program. Accompanying this language is a general-purpose planner CROW, which interprets the behavior description and searches as necessary to find a sound plan. We demonstrate (1) via example programs, that CDL can be used to specify, very intuitively, different known strategies for navigation among movable obstacle (NAMO) problems, (2) via empirical results, that CROW can take advantage of the priors expressed in CDL to very quickly solve problem instances with known simplifying structure but still generalize to hard instances, and (3) via theory, that *width*, a powerful characterization of the worst-case complexity of planning problems, corresponds to a natural property of CDL descriptions and that CROW operates in time on the same order as the width-based worst-case complexity.

Keywords: task and motion planning, hierarchical planning, programming language

1 Introduction

Hybrid discrete-continuous decision problems are critical in robotics. The basic pick-and-place object manipulation is a classic example, where the robot needs to produce a sequence of primitive actions that are parameterized by both discrete variables (objects to manipulate), and continuous parameters (grasping poses, trajectories, etc.). This problem formulation extends to rich problem classes involving non-prehensile manipulation, articulated object manipulation, etc.

In this paper, we consider such hybrid decision problems with discrete-time transitions. As algorithm developers, we must provide an algorithm for the robot to take the current world state (or, in general, a belief state) and produce the next action. Classically there are two main strategies. In *imperative* strategies we construct a direct mapping, in the form of a computer program or neural network weights with no search. Optionally, the mapping can be hierarchical, where a sequence of functions needs to be called as nested subroutines to generate the first action. These approaches use a roughly constant amount of computation (independent of problem size, horizon, etc.) to determine the next action. Typical neural network (hierarchical) policies, behavior trees, PID controllers, etc., fall

(a) The "direct connect" approach.

```
behavior achieve_at_1(g: Pose):
  goal: at(g)
  body:

    bind t: Trajectory where:
      connect(s, g, t)


    forall o: Object:
      achieve not blocking(o, t)
    do exec_traj(t)
```

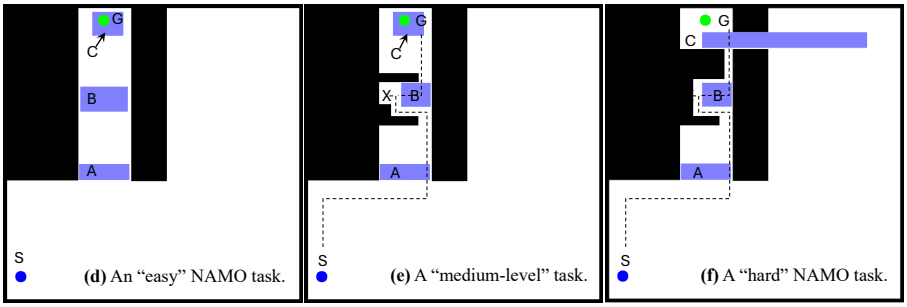
(b) Searching for waypoints.

```
behavior achieve_at_2(g: Pose):
  goal: at(g)
  body:
    bind x: Pose # intermediate pos
    bind t: Trajectory where:
      connect(x, g, t)

    achieve at(x)
    forall o: Object:
      achieve not blocking(o, t)
    do exec_traj(t)
```

(c) Searching for waypoints and orders.

```
behavior achieve_at_3(g: Pose):
  goal: at(g)
  body:
    bind x: Pose
    bind t: Trajectory where:
      connect(x, g, t)
    promotable:
      achieve at(x)
      forall o: Object:
        achieve not blocking(o, t)
    do exec_traj(t)
```



(d) An "easy" NAMO task.

Sol: the robot can drag object A then object B then object C into the empty space to clear a path to reach goal point G.

(e) A "medium-level" task.

Sol: the robot moves to location X (after clearing A), then drags obstacle B downward to clear the path from X to G, then moves to G.

(f) A "hard" NAMO task.

Sol: the robot needs to work at beginning of the trajectory, to enable traversal of the end of the trajectory: it first pulls C out to the right, then drags A into free space to reach X, then moves B down and can reach G.

Fig. 1: (a)–(c):CDL specifications for three different NAMO solutions; (d)–(f): example problems of different degrees of difficulty. Full programs in Appendix C.

into this category. In the second *declarative* strategies, we compute the next action indirectly by first specifying a space of possible solutions, such as all possible sequences of actions and we "search" in this space to find a sequence that satisfies state-dynamics constraints (in terms of a transition model) and that reaches a design goal while minimizing a cost objective. Classical AI planners, task and motion planners, and model-predictive controllers all fall into this category.

There are important trade-offs between these approaches, such as the compactness of the representation and the inference runtime, but they are *not* problem-independent. For some problem classes, there are compact imperative programs that can be directly written by humans or easily learned via imitation or reinforcement learning. At the other extreme, for some problem classes, there is no compact imperative representation (e.g., for PSPACE-Hard problems, see also [20,27]). For such problems, the most effective approach may be to specify or learn a declarative transition model and use it at performance time to search. Some other approaches such as hierarchical task networks (HTNs [24]) and hierarchical planning in the now (HPN [14]) occupy a middle ground between completely imperative and completely declarative methods. They leverage some heuristics to reduce runtime complexity at the cost of some loss in generality.

However, one major drawback of almost all of the existing methods for behavior generation is that they take a *uniform* approach: they are completely imperative or completely declarative or completely follow the restrictions of STRIPS.

But, as we try to address bigger and more general problems (such as designing a general-purpose household assistant robot that can operate over a horizon of days), we will want to "mix and match" these approaches.

The key insight that allows us to unify and combine these approaches is that the differences among them can be described by different types of *branches and commitments*, such as commitments to the order for achieving subgoals, the choice of plans to achieve a particular subgoal, and the possible interleavings of subgoal achievements. A generic formulation for a *behavior* is to think about it as a nondeterministic program that can generate a sequence of control commands to achieve certain goals. Such programs may involve recursive function calls to other behaviors to generate part of the control commands for some "subgoals." In the completely imperative case, we have an action generator that already has all correctness conditions built into it, so that we can immediately commit to a single choice at every turn. In the completely declarative case, we cannot commit to executing any action until we have generated an entire sequence and verified its soundness. In section 4 we detail structural assumptions in the middle ground made by several existing behavior specification methods.

NAMO examples Consider the motivating example of a NAMO (navigation among movable obstacles [25]) domain illustrated in Fig. 1, where the task is to move to a target location potentially moving obstacles in order to reach it. In Fig. 1, we describe a series of solution strategies, using our proposed behavior description language CDL. The core of a CDL program is a set of *behavior rules* that specify methods for achieving some goal. These rules may provide a sequential deterministic program for achieving the goal or may describe a decomposition into one or more abstract or concrete steps without specifying exactly how to do them, in what order, or with what variable bindings.

Fig. 1a describes a hierarchical policy that is almost fully imperative: find a path that connects the current robot position s to the goal g (`bind` statement), move each obstacle out of the way (via another behavior rule, specified in the `achieve` statements), and then execute a controller to traverse the path (`do` statement). This strategy will work for simple problem instances, such Fig. 1d, where the obstacles can be moved individually and there is sufficient additional storage space. Fig. 1b describes a slightly more complex strategy, inspired by Stilman and Kuffner [25]: pick a sequence of waypoints, then traverse to the waypoints, clearing each path segment before it is traversed. Example Fig. 1e, is solvable with this method but not with method in Fig. 1a, because it is not possible to clear the entire path at once. Fig. 1c describes a more general method: like the previous method, it states that we will have to reach a sequence of waypoints, and that the path segments must be cleared, but (via the `promotable` section) it gives the planner the freedom to do these steps in any order, and even to interleave the sub-plans for achieving the subgoals. It solves all problem instances, including Fig. 1f, which cannot be solved by the previous two algorithms. The important observation here is that it is easy, in CDL, to specify a whole range of algorithms in a very clear and concise way.

In summary, our contributions in this paper are:

1. The Crow definition language (CDL), a formalism for specifying robot behavior in hybrid discrete-continuous domains that encompasses imperative and declarative specifications, as well as a highly flexible range of combinations. Furthermore, it enables easy specification (by humans) of domain knowledge that, for example, can reduce the difficulty of a worst-case PSPACE-hard planning problem to a lower-order polynomial by constraining how subtask solutions can be committed independently and serialized.
2. The Crow interpreter for CDL, a behavior-generation algorithm that takes a behavior specification and implements a policy that maps domain states into actions, with an internal computation structure ranging from direct policy-following to complex planning with backtracking and constraint-based variable binding, based on the behavior specification.

2 The Crow Definition Language (CDL)

CDL is a flexible language for describing goal-directed behavior for discrete-time hybrid discrete-continuous space decision-making. At performance time, a CDL interpreter takes a CDL behavior specification, the current state, and the goal as its input. It translates the raw sensory input into a structured representation, then recursively composes behavior rules to find a sequence of control commands that can achieve the goal. Finally, it executes the generated control commands.

A hybrid decision-making problem is a tuple of $\langle \mathcal{S}, \mathcal{C}, \mathcal{T}, s_0, g \rangle$, where $\mathcal{S}$ is the set of states, $\mathcal{C}$ is a set of primitive controllers, $\mathcal{T} : \mathcal{S} \times \mathcal{C} \to \mathcal{S}$ is a (deterministic) transition function, s_0 is the initial state and $g : \mathcal{S} \to \{0, 1\}$ is a goal classifier. It is hybrid because elements in both $\mathcal{S}$ and $\mathcal{C}$ have discrete and continuous values in different "dimensions" (e.g., a set of grasping controllers might be parameterized by $\mathcal{U} \times \mathbb{R}^6$ where $\mathcal{U}$ is the universe of entities in the scene and $\mathbb{R}^6$ denotes grasping poses). In this paper, we only consider tasks where the states are fully observable, transitions are deterministic, and actions are of uniform costs. The task is to find a sequence of primitive controller actions $\bar{a} = \{a_1, \cdots a_T\}$, which are functions that can be executed in the (physical) world. We want the sequence $\bar{a}$ to be *sound* in the sense that $g(s_T)$ is true (goal is reached) and $s_t = \mathcal{T}(s_{t-1}, a_t)$ (state transitions are satisfied) in the world.

Fundamentally, we can view CDL as specifying a *generative model* for sound primitive plans, together with language constructs that can dramatically improve the computational efficiency of finding such plans. The space of plans is generated via non-deterministic choices of methods for achieving subgoals (described in section 2.2), of ordering and interleaving of subgoals (described in section 2.3), and of values for free variables (described in section 2.4). This is generally an infinite space of possible plans, which requires intelligent search. In section 2.2 we show that, by characterizing action effects and asserting conditions that must hold at various points in the plan, we can substantially prune the search space. In section 2.5, we show that, by committing early to some non-deterministic choices, we can get exponential reductions in the space. We include more details and examples in Appendix C and Appendix D.

Interface to perception and control We begin by defining the input and output interfaces between CDL and the robot for which it is specifying behavior.

For planning purposes, *states* ($\mathcal{S}$) are represented in a relational manner, in terms of entities and relational features associated with them. Formally, a state is represented as a tuple $s = \langle U, F \rangle$, where U is the universe of entities, assumed to be a fixed finite set. F is a set of relational features that can take discrete or continuous values. All entities and values are typed in CDL. Each feature $f \in F$ can be viewed as a table where each entry is associated with a tuple of entities $(o_1, \cdots, o_k)$. Each entry has the value of the feature in the state. For example, in NAMO, we define the feature `position(x)` as a unary feature of type `Pose`.

```
typedef Object
typedef Pose: vector[float32, 2]
feature agent_position() -> Pose
feature position(x: Object) -> Pose
feature holding(x: Object) -> bool
```

We assume that an external "perception" system can segment raw sensory input into individual objects and that, for each feature, there is a program defined in an external language (e.g., Python) that can compute the feature value for any tuple of objects of the appropriate arity.

Actual behavior in the world is generated through *controllers*, which are primitive functions that take entities and values as inputs and execute a low-level sensorimotor loop to effect the world, returning control when they have completed. Examples include tracking a motion trajectory or grasping an object.

```
controller exec_traj(t: Trajectory)
controller attach(x: Object)
controller detach(x: Object)
```

Behavior rules The primary part of a CDL program is a set of *behavior rules* $\mathcal{B}$. Primitive plans are generated through a refinement process, which starts with the top-level goal and repeatedly applies behavior rules to a partial plan until it is refined into a completely primitive plan. In CDL, each behavior rule specifies a strategy for achieving the condition that some state feature, applied to some objects, has a desired value (examples might be `holding(o)` or `position(o) == p`). Importantly, they are "lifted" in the sense that the goals they achieve are parameterized by variables. Thus, a behavior rule for `holding(o)` can be used to achieve the holding feature of any object.

Behavior rules play multiple roles in CDL, as we will show in subsequent sections. In their simplest form, each behavior rule is a tuple of $\langle \mathit{name}, \mathit{args}, g, B \rangle$, where *name* is the name, *args* is a set of parameters, g is a goal statement (a Boolean expression over state features) that will be achieved after successfully executing B, and B is the body, which is a non-deterministic program that, through refinement, can generate primitive plans.

2.1 Imperative Behavior Scripting in CDL

The simplest style of behavior specification that CDL supports is deterministic hierarchical policies. In this case, the body $\mathit{body}(b)$ for each behavior rule $b \in \mathcal{B}$

is a deterministic program with subroutines, consisting of a sequence with three types of statements. First,

```
let v = f(v, ...)
```

declares a local variable `v` and assigns a value to it, which is the return value of `f`. f is a function defined in an external language and can take values of the program parameters, and values in previous `let` statements as input. Next,

```
achieve f(...)
```

recursively calls another behavior rule whose goal is the feature `f` applied to variables available in the current rule. Finally,

```
do c(...)
```

inserts into the plan being constructed a call to primitive controller `c` parameterized by some variables in the current rule.

The formal semantics of these statements will be defined later when we describe the plan refinement processes. Here, for illustration purposes, we consider these constructs in the following simple program for clearing a movable obstacle `o` out of the way of a specified trajectory `t`.

```
behavior move_away(o: Object, t: Trajectory)
  goal: not blocking(o, t)
  body:
    # Move the agent to a position where it is close to o
    achieve close_to(o)
    do attach(o)
    # Find a free location for the obstacle o away from t
    let o_pos = free_location(o, t)
    achieve position(o) == o_pos
    do detach(o)
```

This program is fully imperative. Given the target object `o` to move and the trajectory `t` to clear, it begins by calling another rule to achieve the subgoal `close_to(o)`. This will construct the first part of a primitive plan. Then it inserts a call to a primitive controller to `attach(o)`. Next, it has to find a place to put the object (for instance in some designated storage place) using the `let` statement. It calls another rule to generate steps to put the object into position, and adds the final primitive controller to detach the object.

This rule is simple, but can only solve a very limited set of NAMO problems, in which there is plenty of easily accessible storage space so that we do not need to coordinate the placement of objects into it, and where moving each object out of the trajectory does not require moving any other objects.

In the case where $\mathcal{B}$ consists of behavior rules of this type, and where there is at most one $b \in \mathcal{B}$ with the same feature f as its goal, $\mathcal{B}$ deterministically specifies a single primitive plan for an initial goal g. This is effectively the same as a simple type of behavior tree [4]. Such rules can be executed very efficiently,

but they put the entire burden of behavior specification on the programmer. The interpreter *cannot* check the resulting primitive plan for soundness, and relies entirely on the programmer to construct a correct plan.

2.2 Nondeterminsitic Behaviors and Soundness Checking

Fully deterministic behavior rules are hard to write in general, especially for large and complex problems involving many different goals and subgoals spanning over a long horizon. Therefore we transfer some of the burden to the CDL interpreter, by letting the programmer specify a space of possible plans (via non-deterministic behavior rules) and letting the interpreter implement a search to find one that satisfies the goal, as well as possibly other intermediate soundness checks.

There are several types of nondeterminism available in CDL; in this section, we begin with the simplest, in which we allow $\mathcal{B}$ to contain multiple behavior rules with the same goal feature g. For example, there may be other methods for achieving `not blocking(o, t)`, such as sweeping it away with a broom, that require very different action sequences and might have different additional effects (for example, that the robot is holding a broom, or the object is toppled over), which might affect the selection of later steps in the plan.

The programmer now has the freedom to describe many possible action sequences, but we must also characterize, implicitly, which ones will be successful in achieving a desired result. To enable this, we introduce two additional language features: `assert` statements and the `eff` (effect) section. The statement

```
assert e(...)
```

requires that an expression should evaluate to true at a given step in the execution of a plan. The expression `e` can be made up of Boolean combinations of features or be the application of an externally defined function to variable values or features of objects in the rule. The effect section of a behavior rule

```
behavior <name>(<args>):
  eff:
    f[...] = ...
    forall x: Object: g[x, ...]= ...
```

is a short program that describes changes to the state that result from applying the behavior rule. It can contain two types of primitive statements: feature assignments and `for` loops over all objects of a certain type. The value to be assigned can be determined by any external function applied to variable values or features of objects in the rule. Consider the example of a behavior rule that moves objects. It may specify its effects on both the position of the agent and the object. It may begin by asserting a precondition that the robot be holding the object. Including such preconditions is not strictly necessary for correctness, but they have a huge effect on efficiency, as we will see.

Formally, we model the semantics of a set of CDL rules $\mathcal{B}$ in terms of a set of candidate *plan traces* that can be produced by using $\mathcal{B}$ to *refine* an initial goal g. A plan trace is a sequence of three types of statements: `do` statements, `assert` statements, and `eff` statements. The refinement process for goal

g begins with an initial *unrefined plan trace*, which consists of two statements, $\{\texttt{achieve}(g); \texttt{assert}(g)\}_{seq}$. This initial trace will be non-deterministically refined, by using rules in $\mathcal{B}$ to replace `achieve` statements with subsequences of other statements, such as refining an `achieve close_to(o)` with a sequence of primitive movement actions, until we have a fully refined plan trace with only `do`, `assert` and `eff` statements. We define the non-deterministic plan refinement operator `RefineS` as:

function REFINES(P)
  if there are no `achieve` statements in P **then return** P
  else
    Let g, i be the goal and index of the last `achieve` statement in P
    for $b \in \{\mathcal{B} \mid \mathit{goal}(b) = g\}$ **do**
      yield REFINES($P_{<i}$) $\oplus$ REFINES($\mathit{body}(b)$) $\oplus$ $\{\texttt{eff}\ \mathit{eff}(b)\} \oplus P_{>i}$

where $P_{<i}$ denotes the prefix of P preceding index i while $P_{>i}$ denotes the suffix, and $\oplus$ is the sequence concatenation operation. (Note that, in general, we compute a substitution of variable names necessary to match g with $\mathit{goal}(b)$, and apply it to $\mathit{body}(b)$ and $\mathit{eff}(b)$ before adding them to the refined plan trace.) We borrow the "yield" syntax from Python to define a generator function. Each time when the "yield" statement is hit, it produces a plan trace that an outer for-loop such as "**for** $P \in$ REFINES(P_0) **do** ..." can consume.

The refinement operation has the potential to produce a large number of plan traces when some `achieve` statements have multiple possible refinements. Through assertions and effect declarations, $\mathcal{B}$ provides internal *soundness* constraints on primitive plans that can be generated. First, given a plan trace $\bar{a} = \{a_1, a_2, \cdots, a_k\}_{seq}$, and an initial state s_0, at each step $t = 1, 2, \cdots, k$, we define the *projected state* $\hat{s}_t$ as: $\hat{s}_0 = s_0$,

$$\hat{s}_t = \begin{cases} \mathit{update}(\hat{s}_{t-1}, \phi) & \text{if } a_t = \texttt{eff}\ \phi \text{ is the effect of behavior } \phi \\ \hat{s}_{t-1} & \text{otherwise} \end{cases}$$

where $\mathit{update}(\hat{s}, \phi)$ is the result of assigning, in $\hat{s}$, all the feature-value pairs specified in ϕ. A plan trace $\{a_1, a_2, \cdots, a_k\}_{seq}$ is *internally sound* if and only if all asserted conditions evaluate to true at their projected states; that is, for all $t <= k$, if $a_t = \texttt{assert}\ \psi$ then $\mathit{eval}(\psi, \hat{s}_t)$, where $\mathit{eval}(\psi, s)$ is true if the Boolean statement ψ is satisfied in state s. Ultimately, we would like to find plans that are *externally sound with respect to a goal g*; that is, guaranteed to drive the world into a state satisfying g.

Proposition 1. *Internal soundness implies external soundness w.r.t. g when (1) each plan trace ends with a final assertion that g holds; and (2) the effect statements of all behavior rules are consistent with the true world transitions.*

In such cases, executing the sequence of primitive controllers in an internally sound plan trace will achieve the intended goal in the world. Although it may sound difficult to state the effects of all primitive actions, one simple implementation strategy, assuming access to a resettable simulator representing the underlying transition model, is to use the simulator to compute the next state.

2.3 Subgoal Ordering and Promotion

In this section, we enrich the flexibility of the plan generator with two more types of non-determinism: subgoal ordering and interleaving. Consider a situation in which a robot has to place an object in a cupboard: clearly, before it places the object, the cupboard door should be open and the robot should be holding the object. But in which order should it achieve those preconditions? It might depend on how many hands the robot has, the distance between where the object has to be picked up and the cupboard, what kind of handle the cupboard has, etc. It is impossible to pick a good static ordering of these operations and much too difficult for the programmer to write a conditional expression to decide which to do first. So, we can allow the CDL generator to consider different orderings of the subgoals and find one that yields a sound plan. Furthermore, imagine that opening the cupboard requires retrieving a step-stool first. Now, it might turn out that we need to actually interleave the primitive steps for opening the cupboard with those for picking up the object.

To extend CDL to handle these cases, we add three new language features, extend the structure of partially refined plan traces, and provide a more sophisticated refinement algorithm.

Unordered statement groups We allow a behavior rule body to have multiple `unordered` sections, containing `achieve` and `do` statements. This is simply equivalent to having multiple fully ordered rules for the same goal, one for each possible choice of ordering of each unordered group. A common case in which we may wish to have unordered groups is when we wish to achieve some subgoal with respect to all objects of some type. We add a section type `foreach` that provides unordered iteration over those objects. For example,

```
1  unordered:
2    statement_1(); statement_2(); ...
3  foreach o: T:
4    statement_3(o); statement_4(o); ...
```

where ... consists of the same types of statements as in a rule body. Here, the order for statements 1 and 2 are unordered. For each object o of type `T`, the order for executing statements 3 and 4 on them, is also undetermined.

Promotable statement groups Unordered `achieve` statements may be refined into plans in any order, but the corresponding primitive controller actions may not be interleaved. For most planning problems, this is sufficient flexibility, but sometimes more is necessary. To accommodate such problems, we extend the syntax of the body of a behavior rule to allow one set of statements, called a `promotable` section, (typically only `achieve`) to be unordered *and* to have their achievements interleaved.* The promotable section may be preceded and followed by additional ordered or unordered lists of statements of any type.

For example, consider the behavior rule `achieve_at_3` in figure 1(c). The `promotable` section says that the robot has to get to waypoint `x` and ensure that the rest of the trajectory to get from `x` to the goal location `g` is clear. These

* We support statements that are ordered and interleavable, but this is less useful.

things may need to happen in any order (in our example, it has to clear part of the trajectory before moving to `x`; more generally, it might need to further interleave steps for moving the obstacles out of the way).

Plan traces and refinement To generate all plan traces specified by this more expressive class of behavior rules, we adopt a more general view of partially refined plan traces and refinements. We describe this in Appendix A.

First, in general, partially refined plan traces will not be totally ordered. Instead, they will be partially ordered, and represented using nested ordered and unordered sequences. For example, the ordering $\{\{p_1, p_2\}_{seq}, \{p_3, p_4\}_{seq}\}_{unordered}$ states that p_1 will be executed before p_2 and that p_3 will be executed before p_4. However, the relative ordering between, for example, p_1 and p_3 is not specified.

Next, we observe that each rule body can be expressed as $\langle L, M, R\rangle$ with three sections: left (everything before the promotable section), middle (the promotable section), and right (everything after the promotable section). Any of these sections may be empty. In general, the L and R sections may be partially ordered and the M section is unordered. We will adopt this same structure for partially refined plan traces, with the initial plan trace having the form $\langle \emptyset, \emptyset, \{\texttt{achieve}(g), \texttt{assert}(g)\}_{seq}\rangle$.

The generalized plan refinement operator $\mathcal{R}^* : \mathcal{P}^3 \to \mathcal{P}_0$ takes as input a partially refined plan trace and non-deterministically refines it into a plan trace $\bar{a}$. $\mathcal{R}^*(P)$ recursively replaces one of the `achieve` statements in P with the body of a behavior b whose goal matches the achieve statement. To handle possible orders of achieving subgoals, it non-deterministically selects a statement β that could be the last statement in L, M, or R (see Appendix A). If β is not an `achieve` statement, it is directly inserted into the plan. Otherwise, we non-deterministically select a behavior b whose goal matches β to handle multiple behavior rules for goal β.

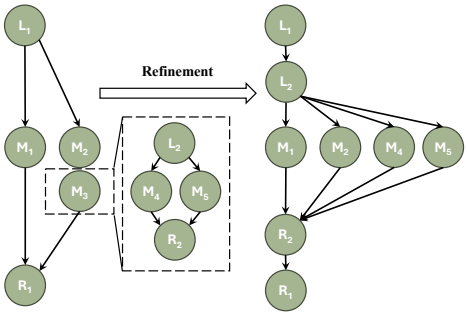


Fig. 2: Illustration of how the refinement operator works. Each node corresponds to a statement (or a sequential program). We first nondeterministically select a node in the "middle" section that is the last statement of one of the sequential chains, and then nondeterministically select a behavior to refine it. Finally, we construct a new plan by "inserting" the refined sub-plan to the graph following the rules.

To handle interleaved execution, we consider two cases: when β is from M and when β is not. Let $\langle L', M', R'\rangle$ be the three sections of $body(b)$. When β is not from M, we simply refine $\langle L', M', R'\rangle$ recursively and concatenate the resulting plans. When β is from M, statements in L' will be appended to L, those in R' will be prepended to R, and M will be unioned with the rest of M (i.e., $M \setminus \{\beta\}$, where the $\setminus$ operator (set difference) returns a partially ordered plan just as M but with β removed).

Figure 2 illustrates the operation on the partially ordered graph. The key insight here is that if q is a subgoal in a partially ordered plan β, we do not assume knowledge of how M' and the rest of the subgoals in $M \setminus \{\beta\}$ can be ordered.

This new rule for promotable sections is inspired by concepts from hierarchical planning and *subgoal serialization*. In conventional hierarchical task and goal networks [22,1], the "bodies" from two different behaviors are always unordered. This is equivalent to marking all statements as promotable. In contrast, planning can be exponentially more efficient when subgoals can be serialized [20], which is the behavior of the "left" and the "right" sections. CDL unifies these two types of behavior rules allowing the programmer fine-grained control to obtain just enough generality while retaining as much efficiency as possible.

2.4 Variable Binding

In the examples we have seen so far, we have had variables that are deterministically bound using `let` statements. In most robotics domains, however, we need to choose values from large or infinite sets (of configurations, grasps, poses, trajectories, etc.) and a crucial part of the planning process is selecting these values to construct a sound plan.

Some task and motion planning systems view the problem of variable binding as one of constraint satsifaction [10]. We adopt this view, allowing a behavior rule to indicate that the value of a variable can be chosen non-deterministically, possibly subject to a constraint that relates its value to the values of other variables. We introduce a new statement type:

```
1  bind v: T = f(v, ...)
```

which requires that `v` have a value v of type `T` that satisfies a condition `f`, expressed as a function (defined in an external language) of the value v, values of the program parameters, and values in previous `bind` statements. We have an example in the behavior rule `achieve_at_3` in figure 1(c): The first of `bind` statement for `x` only constrains the variable by its type. The second one for trajectories can be reduced to two statements:

```
1  bind t: Trajectory; assert connect(x, g, t)
```

However, for the purposes of solving the embedded constraint-satisfaction problem, it is useful to keep the variable-value constraints separate. It is also important to note that in general, constraint functions should be accompanied by conditional generators that can produce satisfying values of the variable being bound given values for other arguments. In our example, a motion planner could be used to generate possible trajectories between a given pair of `x` and `g`.

When we have `bind` statements, applying the refinement operator on an initial unrefined plan trace will give a set of variable-augmented plan traces. Let the $\langle \bar{a}, V \rangle$ be a tuple of the plan trace $\bar{a}$ with a set of associated variables V. $\langle \bar{a}, V \rangle$ is *sound* if and only if there exists an assignment A (mapping from variables in V to their domains) where $\bar{a}_{V \setminus A}$ is a sound plan. Here, the $\bar{a}_{V \setminus A}$ operation replaces all variables in $\bar{a}$ with their assignments in V.

The ordering of `bind` statements in the final plan suggests a variable ordering for solving the CSP at plan-generation time, although any sound (and ideally complete) CSP-solving method can be used. We include details in Appendix B.

2.5 Commitment

In some cases, there may be many alternative solutions for a subpart of a problem, any of which will work fine with any solution for the next part. In such cases, it may be very expensive at planning time to maintain all of those alternatives. To allow a programmer to communicate this situation to the interpreter algorithm, we add a `commit` statement, inspired by Prolog's `cut`, to CDL. It does not increase the space of possible plans or change the soundness conditions. If used incorrectly, however, it can endanger completeness by prematurely ruling out a potentially good alternative. To gain intuition, consider a modification to part of the behavior rule `achieve_at_1` in figure 1(a) for moving obstacles:

```
forall o: Object:
  achieve not blocking(o, t)
  commit # newly added!
```

Here, we have added a `commit` statement, so that after finding a path for clearing some object $o1$ from the trajectory `t` the interpreter will commit to the first part of the plan trace as well as the continuous variables associated with it (e.g., the new location for the object `o1`). Adding this explicit commit forbids trying different placements for this object, but does not forbid placing the objects in a different order. When there is enough space for reorganizing objects, this behavior can succeed with high probability while significantly improving runtime.

A `commit` statement can appear in the left or right section of a behavior program. Take the left section as an example. When we refine $\langle \{...; \texttt{commit}\}_{seq}, M, R\rangle$, instead of considering all possible refinements to the rest of the left program $\textsc{refine}(\langle L \setminus \{\beta\}, \emptyset, \emptyset\rangle)$, we instead consider an arbitrary one of them. In practice, this may be implemented by considering only the first plan trace generated from the recursive refinement to possibly prune out a large number of branches — we will see this in action in our CROW interpreter in Section 3.

3 The CROW Interpreter

Next, we describe a practical implementation of an interpreter for CDL, called CROW (Compositional Regression and Optimization Wayfinder). It is a nondeterministic algorithm, with a structure similar to the refinement operator, but crucially with pruning based on assertions and commitment.

It keeps track of a program tuple $\langle L, M, R\rangle$, and a set of variables and constraints $\langle V, C\rangle$. It uses the following implementations to handle behavior branching and continuous constraint satisfaction — it is a depth-first search algorithm given a set of behaviors $\mathcal{B}$. Its basic structure is to take the program tuple as input and recursively compute an output tuple consisting of a resulting state s', action sequence $\bar{s}$, variable set V, and constraint set C.

1. If L is not empty, it recursively solves $\langle \emptyset, \emptyset, L \rangle$ first, and then solves $\langle \emptyset, M, R \rangle$.
2. Else, if M is not empty, it recursively solves M and then R. When it is solving M, it will consider merging promotable sections of the new behavior and the current M following the definition of the refinement operators.
3. Else, it solves statements in R one by one and concatenates the plan traces.

It defers the constraint satisfaction solving after checking all purely discrete constraints by tracking variable ordering throughout the expansion.

---

function PLAN($state$, g, $\mathcal{B}$) ▷ Top-level entry
  let $P_0 \leftarrow \langle \emptyset, \emptyset, \{\texttt{achieve}(g); \texttt{assert}(g)\}_{seq}, \emptyset, \emptyset \rangle$
  for $\langle s, \bar{a}, V, C \rangle \in$ CROW($state, P_0, \mathcal{B}$) **do**
    if SOLVE-CSP(V, C) returns an assignment A **then**
      return $\bar{a}_{V \setminus A}$ ▷ We have found a sound plan
function CROW(s, $\langle L, M, R, V, C \rangle$, $\mathcal{B}$)
  if $L \neq \emptyset$ **then**
    for $\langle s', \bar{a}, V', C' \rangle \in$ CROW($s, \langle \emptyset, \emptyset, L, V, C \rangle, \mathcal{B}$) **do**
      for $\langle s'', \bar{a}', V'', C'' \rangle \in$ CROW($s', \langle \emptyset, M, R, V, C \rangle, \mathcal{B}$) **do**
        yield $\langle s'', \bar{a} \oplus \bar{a}', V'', C'' \rangle$
  else if $M \neq \emptyset$ **then**
    for $\beta \in \mathit{tail}(M)$ **do**
      if β is `achieve` **then**
        for $b \in \{\langle L', M', R' \rangle \in \mathcal{B} \mid \mathit{goal}(b) = \beta\}$ **do**
          let $P' = \langle \{L, L'\}_{seq}, \{M \setminus \{\beta\}, M'\}_{unordered}, \{R', R\}_{seq}, V, C \rangle$
          yield from CROW($s, P', \mathcal{B}$)
      else yield from CROW($s, \langle L, M \setminus \{\beta\}, \{\beta, R\}_{seq}, V, C \rangle, \mathcal{B}$)
  else if $R \neq \emptyset$ **then**
    for $\beta \in \mathit{tail}(R)$ **do**
      for $\langle s', \bar{a}, V', C' \rangle \in$ CROW($s, \langle L, M, R \setminus \{\beta\}, V, C \rangle, \mathcal{B}$) **do**
        if β is `assert` **then**
          if β evaluates deterministically **then**
            if $\beta(s'')$ **then yield** $\langle s', \bar{a}, V', C' \rangle$ ▷ Yield only if $\beta(s'')$ is true
          else yield $\langle s', \bar{a}, V', C' \cup \{\beta\} \rangle$ ▷ Defer β check to CSP
        else if β is `commit` **then**
          if SOLVE-CSP(V', C') returns an assignment A **then**
            yield $\langle s'_{V \setminus A}, \bar{a}_{V \setminus A}, \emptyset, \emptyset \rangle$; **break** ▷ Yield only the first plan trace
        else if β is `achieve` **then**
          for $b = \langle L', M', R' \rangle \in \mathcal{B} \mid \mathit{goal}(b) = \beta$ **do**
            yield from CROW($s', \langle L', M', R', V', C' \rangle, \mathcal{B}$)
        else if β is `do` **then yield** $\langle s', \bar{a} \oplus \{\beta\}, V', C' \rangle$
        else if β is `bind` **then return** $\langle s', \bar{a}, V' \cup \{\mathit{var}(\beta)\}, C' \cup \{\mathit{body}(\beta)\} \rangle$
        else if β is `effect` **then return** $\langle \mathit{update}(s', \beta), \bar{a}, V', C' \rangle$

The function CROW defined above is not a good control structure, as it effectively does a left-branching depth-first traversal of the space of plans. To make it practical, we wrap it in an iterative-deepening depth-first search (IDDFS), which searches with an increasing depth limit until a sound plan is found.

Theorem 1. *Suppose there exists a sound plan that can be generated by the behavior specification $\mathcal{B}$ at a finite depth of nested* `achieve` *statements. Then the IDDFS* CROW *will always terminate and return a sound plan.*

(Proof in appendix A.1). While the IDDFS CROW algorithm is easy to implement and comprehend, other implementations such as breadth-first search or A-Star search with heuristics, would also be possible.

3.1 Problem Hardness and CROW Runtime

In discrete-space planning without `bind` statements, based on languages like PDDL, it is typical to be completely general and put all `achieve` statements into the `promotable` section, which can result in poor worst-case runtimes. However, if we know that there are some subgoals whose achievement is independent of others, then we can put all and only the `achieve` statements that could depend on each other into the promotable section, and for the other, independent subgoals, commit after achieving each one. In such cases, an upper bound on the problem complexity is related to the size of the largest $\langle L, M, R\rangle$ tuple during the refinement process. Here, we restate the problem hardness results from STRIPS planning [2] in the language of CDL.

Theorem 2 (Mao et al. [20]). *Consider the refinement process applied to an initial program P_0 and an initial state s_0, given a set of behaviors $\mathcal{B}$. If there is a sound plan that can be refined by allowing maximally K* `achieve` *statements in $M + R$ of any partially refined plan traces $\langle L, M, R\rangle$, the complexity of finding a sound plan is upper-bounded by $N^{O(K)}$ where N is the number of entities in the environment, and K is called the "width" of the planning problem.*

Proof sketch. For all possible programs p, let *subgoals*(p) be the set of `achieve` statements inside p. No matter the order of elements in p, all we care about is finding a feasible plan that simultaneously achieves all subgoals in *subgoals*(p), and there are $N^{O(K)}$ possible subsets. Thus, it can be solved in at most $N^{O(K)}$.

This result gives us a fine-grained insight into a useful class of planning problems in terms of their hardness, which essentially depends on the total number of subgoals that accumulate in the program, usually in the "middle" section. Mao et al. [20] also proved that, for many practical problems, K is a small constant usually independent of the number of entities in the environment. For practitioners, this suggests an alternative search algorithm for implementing a CDL interpreter, which is an iterative-widening algorithm [18], where, instead of iterating over the depth of the refinement process, we iterate over the maximal number of `achieve` statements allowed to accumulate during the refinement process. In this case, the iterative-widening (CROW-IW) algorithm will be able to solve the problem in time bounded by $N^{O(K)}$. This property of width also helps us gain insights

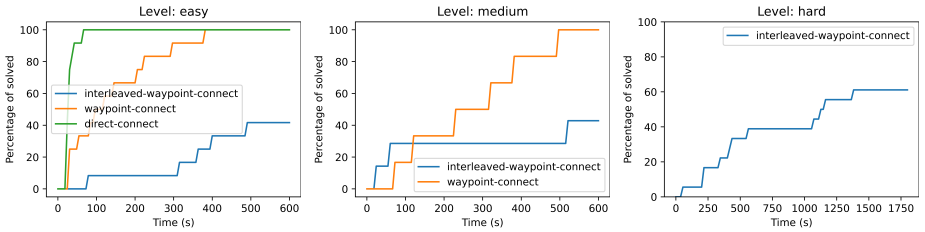


Fig. 3: Percentage of successful executions as a function of wall-clock seconds; each plot corresponds to a NAMO problem instance, and each line-color corresponds to a behavior program.

into the CSP solving hardness in CROW. Essentially, if we hypothesize that the amount of time needed for solving continuous CSPs is exponential in the number of variables, we would obtain a complexity of CROW on generic hybrid decision-making problems which is a polynomial whose exponent is linear in the program width and the number of variables accumulated in CSPs (due to non-`commit`). This hypothesis could be difficult to satisfy in problems where generating values that satisfy the constraints (e.g. by sampling) is hard.

3.2 Experiments in NAMO

In the introduction, we used the NAMO problem to illustrate the flexibility of CDL for expressing different spaces of possible plans, and argued that this flexibility is useful because a general-purpose planning algorithm can take advantage of reduced plan spaces without requiring special-purpose algorithms to be implemented. Here, we explore the effectiveness of the CDL and our interpreter on those three CDL behavior specifications: the *direct-connect* approach which first plans the path from the initial position to the goal and then clears all obstacles it hits; the *waypoint-connect* approach which searches for a set of intermediate waypoints, and plans for the whole path by clearing all obstacles between each consecutive waypoint in a stagewise manner; and the *interleaved-waypoint-connect* approach which considers all possible orders for the movements and obstacle-clearing. Intuitively, these three algorithms are ordered in a way that the later algorithms can solve a large set of problems than previous ones.

We consider the three example NAMO problems presented in Fig. 1 and apply the three behavior programs to these problems, and show the results in Figure 3. Since there is substantial nondeterminism in the algorithm, we run CROW on each behavior program 10 times for each problem. The results match our intuition of the benefits of providing a behavior specification appropriate to the problem class. Overall, if we apply a more advanced strategy for solving "simpler" problems (e.g., using the waypoint-connect algorithm for the easy-level NAMO tasks), due to the additional branching factors it considers (for example, the direct-connect approach does not consider waypoints while the waypoint-connect algorithm will try sampling different waypoints), the overall runtime is longer than for the methods specifically tailored towards the scenario, although given a sufficient amount of time, they are capable of solving the problem.

Importantly, we did not have to write any new code to get these results: the general CROW algorithm running on the simple alternative strategies in Fig. 1 is all that was necessary, and it would be easy to further explore a large the trade-off space of behavior specifications, generality, and efficiency.

4 Related Work and Discussion

CDL builds on a series of insights from hierarchical planning [1], subgoal serialization [18,20], and constraint satisfaction-based hybrid discrete-continuous planning [10,9]. Its semantics allow for the specification of a large body of behavior types. For example, behavior trees [4] can be described by CDL behaviors without effect sections. Preconditions of rules can be written as assert statements at the top of the body. A classical STRIPS operator definition [8] is equivalent to a CDL behavior with all preconditions written in its promotable section (but note that the original STRIPS *implementation* had all preconditions `unordrered` but not `promotable`, making it incomplete). Furthermore, any atomic STRIPS effect statements [17] can also be written as assignment statements in CDL. Therefore, it is one of the behavior types with the weakest commitments. In this case, without any further control structure, the interpreter CROW is sound and complete but very inefficient. This could be improved by adding heuristics [13,12]. The most important benefit of CDL is that when programmers know what can be serialized, they can use CDL to declare those commits to make search efficient.

The most closely-related planning frameworks are the hierarchical task and goal networks [7,1], where an operator is equivalent to a CDL behavior with an ordered promotable action body containing preconditions: they are always ordered but interleavable. However, being a uniform approach that semantically always allows for interleaved execution of subgoal routines, they support neither serialization nor commits. All these behavior specifications can be implemented in CDL, as well as combined flexibly to address non-homogenous problems.

The idea of explicitly separating problem-solving into generators and tests dates back Newell and Simon [23], and was more recently applied to robot programming and planning starting with GOLOG [16], its variants [5,6], and Planning with Loops [15]. However, their generators are specified in a generic programming language where interleaved behaviors, commits, and constraints need to be programmed manually. CDL has the same spirit as these algorithms but is built on insights from hierarchical planning, subgoal serialization, and constraint-based hybrid planning to define a set of language supports.

Future work We have discussed at length the strengths of CDL and CROW as a framework for *planning*; but it also exposes a set of opportunities for *learning*. There is existing work for learning both transition models and policies [3,19] for hybrid domains: CDL offers a formalism for integrating learned partial models and policies as compositional behavior rules. In addition, we have demonstrated the critical role of ordering, assertions, and commitment in making planning efficient. A completely new learning opportunity is to discover and incorporate this structure. In addition, although there is a growing body of work in learning individual samplers for TAMP systems [21,28,11], CDL exposes the opportunity to also learn variable ordering and commitment strategies.

Acknowledgements. We gratefully acknowledge support from NSF grant 2214 177; from AFOSR grant FA9550-22-1-0249; from ONR MURI grant N00014-22-1-2740; and from ARO grant W911NF-23-1-0034; from MIT Quest for Intelligence; from the MIT-IBM Watson AI Lab; from the Boston Dynamics AI Institute; from ONR Science of AI; and from Simons Center for the Social Brain. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

1. Alford, R., Shivashankar, V., Roberts, M., Frank, J., Aha, D.W.: Hierarchical Planning: Relating Task and Goal Decomposition with Task Sharing. In: IJCAI (2016) 11, 16
2. Bylander, T.: The Computational Complexity of Propositional STRIPS Planning. Artif. Intell. **69**(1-2), 165–204 (1994) 14
3. Chitnis, R., Silver, T., Tenenbaum, J.B., Lozano-Perez, T., Kaelbling, L.P.: Learning Neuro-Symbolic Relational Transition Models for Bilevel Planning. In: IROS (2022) 16
4. Colledanchise, M., Ögren, P.: Behavior Trees in Robotics and AI: An Introduction. CRC Press (2018) 6, 16
5. De Giacomo, G., Lespérance, Y., Levesque, H.J.: ConGolog: A Concurrent Programming Language Based on the Situation Calculus. AIJ **121**(1-2), 109–169 (2000) 16
6. De Giacomo, G., Lespérance, Y., Levesque, H.J., Sardina, S.: IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents. Multi-Agent Programming: Languages, Tools and Applications pp. 31–72 (2009) 16
7. Erol, K., Hendler, J.A., Nau, D.S.: Semantics for Hierarchical Task-Network Planning. University of Maryland College Park (1994) 16
8. Fikes, R.E., Nilsson, N.J.: STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. Artif. Intell. **2**(3-4), 189–208 (1971) 16
9. Garrett, C.R., Chitnis, R., Holladay, R., Kim, B., Silver, T., Kaelbling, L.P., Lozano-Pérez, T.: Integrated Task and Motion Planning. Annual Review of Control, Robotics, and Autonomous Systems **4**(1), 265–293 (2021) 16
10. Garrett, C.R., Lozano-Pérez, T., Kaelbling, L.P.: Sampling-Based Methods for Factored Task and Motion Planning. IJRR **37**(13-14), 1796–1825 (2018) 11, 16, 22
11. Ha, J.S., Driess, D., Toussaint, M.: Deep Visual Constraints: Neural Implicit Models for Manipulation Planning from Visual Input. RA-L (2022) 16
12. Helmert, M.: The Fast Downward Planning System. JAIR **26**, 191–246 (2006) 16
13. Hoffmann, J., Nebel, B.: The FF Planning System: Fast Plan Generation through Heuristic Search. JAIR **14**, 253–302 (2001) 16
14. Kaelbling, L.P., Lozano-Pérez, T.: Hierarchical Task and Motion Planning in the Now. In: ICRA (2011) 2
15. Levesque, H.J.: Planning with Loops. In: IJCAI. pp. 509–515 (2005) 16
16. Levesque, H.J., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.B.: GOLOG: A Logic Programming Language for Dynamic Domains. The Journal of Logic Programming **31**(1-3), 59–83 (1997) 16
17. Lifschitz, V.: On the semantics of STRIPS. In: Workshop on Reasoning about Actions and Plans (1987) 16

18. Lipovetzky, N., Geffner, H.: Width and Serialization of Classical Planning Problems. In: ECAI (2012) 14, 16
19. Mao, J., Lozano-Pérez, T., Tenenbaum, J., Kaelbling, L.: PDSketch: Integrated Domain Programming, Learning, and Planning. In: NeurIPS (2022) 16
20. Mao, J., Lozano-Pérez, T., Tenenbaum, J., Kaelbling, L.: What Planning Problems Can A Relational Neural Network Solve? In: NeurIPS (2023) 2, 11, 14, 16
21. Mendez-Mendez, J., Kaelbling, L.P., Lozano-Pérez, T.: Embodied Lifelong Learning for Task and Motion Planning. In: CoRL. PMLR (2023) 16
22. Nau, D.S., Au, T.C., Ilghami, O., Kuter, U., Murdock, J.W., Wu, D., Yaman, F.: SHOP2: An HTN Planning System. JAIR **20**, 379–404 (2003) 11
23. Newell, A., Simon, H.A., et al.: Human Problem Solving, vol. 104. Prentice-hall Englewood Cliffs, NJ (1972) 16
24. Sacerdoti, E.D.: The Nonlinear Nature of Plans (1975) 2
25. Stilman, M., Kuffner, J.J.: Navigation Among Movable Obstacles: Real-Time Reasoning in Complex Environments. International Journal of Humanoid Robotics **2**(04), 479–503 (2005) 3
26. Sussman, G.J.: A Computational Model of Skill Acquisition. AI Technical Reports (1973) 35
27. Vega-Brown, W., Roy, N.: Task and Motion Planning is PSPACE-Complete. In: AAAI (2020) 2
28. Yang, Z., Mao, J., Du, Y., Wu, J., Tenenbaum, J.B., Lozano-Pérez, T., Kaelbling, L.P.: Compositional Diffusion-Based Continuous Constraint Solvers. In: CoRL (2023) 16

A Refinements and The Crow Interpreter

In this section, we formally define the refinement rules for CDL behavior rules. We start with a definition of types of statement orders.

Statement orders For a set of statements, we define two basic orders.

1. Sequential program, represented as an ordered set $\{p_1, p_2, \cdots, p_k\}_{seq}$. Intuitively, all statements will be executed in this order.
2. Unordered program, represented as an unordered set $\{p_1, p_2, \cdots, p_k\}_{unordered}$. Intuitively, all statements can be executed in arbitrary orders (there are in total $k!$ possible serialization of the program).

These two types of primitive ordering can be nested, to represent partial orders. For example, the ordering $\{\{p_1, p_2\}_{seq}, \{p_3, p_4\}_{seq}\}_{unordered}$ states that p_1 will be executed before p_2 and that p_3 will be executed before p_4. However, the relative ordering between, for example, p_1 and p_3 is not specified. In general, we will call these programs *partially ordered programs*. The set of all possible partially ordered plans (including sequential ones and unordered ones) is denoted as $\mathcal{P}$.

As an example, consider the following `foreach` statement.

```
foreach o: T:
  statement_1(o); statement_2(o); ...
```

In a domain with two objects, it would be effectively equivalent to

```
unordered:
  sequential:
    statement_1(o1); statement_2(o1); ...
  sequential:
    statement_1(o2); statement_2(o2); ...
```

Next, we define the *tail* operator that takes a partially ordered program p and returns a set of statements in p that might be the "last" statement in one of the serializations of p. It can be defined recursively:

$$tail(p) = \begin{cases} \{p\} & \text{if } p \text{ is a single subgoal or a controller call} \\ tail(p_k) & \text{if } p = \{p_1, \cdots, p_k\}_{seq} \text{ is a totally ordered plan} \\ \bigcup_{p_i \in p} tail(p_i) & \text{if } p \text{ is an unordered plan} \end{cases}.$$

One can iteratively select an element from $tail(p)$ nondeterministically to construct all possible serializations of p. In CDL, there are two keywords `sequential` and `unordered` for specifying a particular section of the program being sequential or unordered. In CDL, a behavior body can contain unordered sections of `achieve` statements, indicating a scenario where the order for achieving those subgoals is unspecified by the programmer, but rather, should be determined using search.

The generalized refinement operator Each behavior body is a program $P = \langle L, M, R \rangle$ with three sections: left (everything before the promotable section), middle (the promotable section), and right (everything after the promotable

section). The generalized plan refinement operator $\mathcal{R}^* : \mathcal{P}^3 \to \mathcal{P}_0$ takes as input a program tuple $P = \langle L, M, R\rangle$ and nondeterministically refines it into a plan trace $\bar{a}$. $\mathcal{R}^*(P)$ iteratively replaces one of the `achieve` statements in P with the body of a behavior b whose goal matches the achieve statement.

Next, to handle possible orders of achieving subgoals, it non-deterministically selects a statement β from $\mathit{tail}(L)$, $\mathit{tail}(M)$, or $\mathit{tail}(R)$. If β is a non-achieve statement, it is directly appended to the plan. Otherwise, we non-deterministically select a behavior b that matches β to handle multiple behavior rules for achieving the goal β.

Finally, to handle interleaved execution, we consider two cases: when β is from M and when β is not. Let $\langle L', M', R'\rangle$ be the three sections of $\mathit{body}(b)$. When β is not from M, we simply refine $\langle L', M', R'\rangle$ recursively and stitches together the outputs. When β is from M. Informally, statements in L' will be appended to L, R' will be prepended to R, and M will be mixed with the rest part of M (i.e., $M \setminus \{\beta\}$, where the $\setminus$ operator (set difference) returns a partially ordered plan just as M but with β removed). The following program shows the program that defines REFINE operator algorithmicly.

function REFINE($P = \langle L, M, R\rangle$)
  for β nondeterministically from $\mathit{tail}(L) \cup \mathit{tail}(M) \cup \mathit{tail}(R)$ **do**
    if $\beta \in \mathit{tail}(L)$ **then**
      for $\bar{a}_1 \in$ REFINE($L \setminus \{\beta\}, \emptyset, \emptyset$), $\bar{a}_2 \in$ REFINE($\emptyset, M, R$) **do**
        if β is `achieve` **then**
          for $b \in \mathcal{B} \mid \mathit{goal}(b) = \beta$ **do yield** $\bar{a}_1 \oplus$ REFINE(b) $\oplus\, \bar{a}_2$
        else yield $\bar{a}_1 \oplus \{\beta\}_{seq} \oplus \bar{a}_2$
    else if $\beta \in \mathit{tail}(R)$ **then** ▷ Similarly to the case of $\beta \in \mathit{tail}(L)$
      for $\bar{a}_1 \in$ REFINE($L, M, \emptyset$), $\bar{a}_2 \in$ REFINE($\emptyset, \emptyset R \setminus \{\beta\}$) **do**
        if β is `achieve` **then**
          for $b \in \mathcal{B} \mid \mathit{goal}(b) = \beta$ **do yield** $\bar{a}_1 \oplus$ REFINE(b) $\oplus\, \bar{a}_2$
        else yield $\bar{a}_1 \oplus \{\beta\}_{seq} \oplus \bar{a}_2$
    else if $\beta \in \mathit{tail}(M)$ **then**
      if β is `achieve` **then**
        for $b = \langle L', M', R'\rangle \in \mathcal{B} \mid \mathit{goal}(b) = \beta$ **do**
          yield REFINE($\{L, L'\}_{seq}, \{M \setminus \{\beta\}, M'\}_{unordered}, \{R', R\}_{seq}$)
      else
        for $\bar{a}_1 \in$ REFINE($L, M \setminus \{\beta\}, \emptyset$), $\bar{a}_2 \in$ REFINE($\emptyset, \emptyset, R$) **do**
          yield $\bar{a}_1 \oplus \{\beta\}_{seq} \oplus \bar{a}_2$

It is important to observe that CDL separates the definition of the generative model for plan traces in the behavior rule bodies from the definition of preconditions and effects of taking actions in the world in the assertions and effects clauses of behavior rules. This allows the behavior specification to be separated into two models: $\mathcal{G}$, a possible plan generator, and $\mathcal{T}$, a plan soundness tester. The task of a CDL interpreter is to find a plan $P \in \mathcal{G}$ such that $\mathcal{T}(P)$. The most naïve implementation would be to enumerate $P \in \mathcal{G}$ in some order and return the first plan P where $\mathcal{T}(P)$ is true. This method would generally be very inef-

ficient, or even incomplete in some continuous domains. In section 3 we provide an algorithm that leverages soundness checks on partially refined plan traces, as well as other mechanisms to dramatically improve efficiency. A more advanced algorithm may leverage partially refined plans from $\mathcal{G}$ (recall that the plan is generated hierarchically), and run tests on part of the plan, to be more efficient. Other methods such as leveraging heuristics or optimizing expansion ordering are also compatible.

A.1 The Completeness and Soundness of Crow

Based on the algorithmic definition of refinement operators, we can now prove the soundness of the Crow interpreter, as stated in Theorem 1.

Proof. The high-level idea is to prove that, under a given depth, Crow will enumerate all possible plan traces that can be generated by the Refine operator. Then, since we are iteratively increasing the allowed depth for the algorithm it is guaranteed that a sound plan will be generated.

We now prove the completeness of the enumerate strategy that Crow employs. Given a particular program $\langle L, M, R\rangle$, we show that $\textsc{Refine}(\langle L, M, R\rangle)$ can be decomposed into the refinement of three programs L, M, and R independently. Formally:

$$\textsc{Refine}(\langle L, M, R\rangle) = \{\bar{a}_1 \oplus \bar{a}_2 \oplus \bar{a}_3 \mid \bar{a}_1 \in \textsc{Refine}(\langle L, \emptyset, \emptyset\rangle), \bar{a}_2 \in \textsc{Refine}(\langle \emptyset, M, \emptyset\rangle), \bar{a}_3 \in \textsc{Refine}(\langle \emptyset, \emptyset, R\rangle)\}$$

For convenience, we define

$$\textsc{Refine}(P) \times \textsc{Refine}(Q) = \{\bar{a}_1 \oplus \bar{a}_2 \mid \bar{a}_1 \in \textsc{Refine}(P), \bar{a}_2 \in \textsc{Refine}(Q)\}$$

This can be proved by induction. First, when $L = M = R = \emptyset$ this is correct. Now consider for any $\beta \in \mathit{tail}(L) \cup \mathit{tail}(M) \cup \mathit{tail}(R)$:

1. If $\beta \in \mathit{tail}(L)$, the refinement is already explicitly decomposed into the refinement of $\textsc{Refine}(\langle L, \emptyset, \emptyset\rangle) \times \textsc{Refine}(\langle \emptyset, M, R\rangle)$, which, by induction is equal to $\textsc{Refine}(\langle L, \emptyset, \emptyset\rangle) \times \textsc{Refine}(\langle \emptyset, M, \emptyset\rangle) \times \textsc{Refine}(\langle \emptyset, \emptyset, R\rangle)$.
2. If $\beta \in \mathit{tail}(R)$, the refinement is symmetric to the L case.
3. If $\beta \in \mathit{tail}(M)$, we are recursively refining the following program: $\langle \{L, L'\}_{seq}, \{M \setminus \{\beta\}, M'\}_{unordered}, \{R', R\}_{seq}\rangle$. For a sequential program $\{L, L'\}_{seq}$, its refinement has the property that $\textsc{Refine}(\{L, L'\}_{seq}) = \textsc{Refine}(L) \times \textsc{Refine}(L')$ by the definition of *tail*. Therefore, this refinement is equivalent to: $\textsc{Refine}(L) \times \textsc{Refine}(L') \times \textsc{Refine}(\{M \setminus \{\beta\}, M'\}_{unordered}) \times \textsc{Refine}(R') \times \textsc{Refine}(R) = \textsc{Refine}(\langle L, \emptyset, \emptyset\rangle) \times \textsc{Refine}(\langle \emptyset, M, \emptyset\rangle) \times \textsc{Refine}(\langle \emptyset, \emptyset, R\rangle)$

Therefore, the Crow left-most refinement strategy is complete. And it is sound because it all assertions have been explicitly tested either during the refinement process, or during the CSP solving.

B Formulations and Algorithms for Constraint Satisfaction Problems

A constraint satisfaction problem is a tuple of $\langle V, C \langle$, where V is the set of variables and C is a set of constraints. CSP solving focuses on finding a satisficing solution to all variables in V that satisfies all constraints in C. Each constraint $c \in C$ is a function that takes variables in V as inputs and returns a Boolean value.

The caveat of using the CSP formulation in decision-making is that the constraints are defined alongside the plan refinement processes. Therefore, depending on the particular time t when specific constraints are evaluated and added to the constraint set, they will have the state s_t implicitly as inputs to those constraints functions. In this case, these functions are no-longer pure functions in the sense that they depend on additional information other than its input.

There are multiple ways to handle this. In our implementation, we handle this using situation calculus. In particular, each feature $f(...)$ should now be considered as a fluent $f_t(...)$. For example, consider we have the following assignment statement at step t of a feature $f(o_1, o_2)$.

```
f[o_1, o_2] = g(arg1, arg2)
```

In this case, if `arg1` or `arg2` are free variables, the evaluation result of the function g will also be a free variable (e.g., `f_t_o1_o2`) which satisfies the following constraint:

```
assert f_t_o1_o2 = g(arg1, arg2)
```

We describe such translations in detail in Appendix D.5.

Using this translation, all we have is a set collection of constraints that are pure, which can be solved by a CSP solver.

B.1 Sampling-based Continuous Constraint Satisfaction Problem Solver

In Crow, we implemented a solver based on sampling from generators following Garrett et al. [10]. It is a depth-first-search-based solver using the DPLL heuristic for acceleration. At a high level, at each step, it first searches for variables whose values can be fully determined based on the current set of assignments A. This includes three cases:

1. The constraint is Boolean constraint, such as `a and b and c and ...`. Then we immediately know that all variables `a, b, c, ...` must be true. If it is `a or b or c or ...` and all variables except for `a` has been assigned to false, then we know `a` must be true.
2. The constraint is an equality constraint of type `f(...) = rv` and all the arguments to the function `f` have been determined. In this case, we can deterministically evaluate `f` to get the assignment to `rv`.

3. There is an equality constraint `f(...) = rv` but the return value `rv` is never appeared in other constraints except for this equality constraint. We can ignore this constraint (and `rv`).

If none of these cases are true, we consider performing a search over the free variables. We prioritize the search for discrete variables (e.g., Boolean values). We essentially nondeterministically choose an assignment to the variable and continue the search for other variables, and backtrack if we can not find assignments to other variables.

When there is no remaining discrete variables, we consider sampling for continuous variables (e.g., grasps, poses, and trajectories). In this case, we will iterate over all constraints and find constraints that matches a user-provided generator. We will describe the detailed syntax for generators in Appendix D but in short, a generator is defined using the following syntax.

```
generator gen_f(x: T, y: T, ...):
  goal: f(x, ...)= y
  in: y, ...
  out: x
```

Here we have defined a generator associated with the function `f`. Given the value of `y` and other arguments to the function `f`, this generator will generate a set of values for the variable `x` such that `f(x, ...) = y` evaluates to true. If we locate a constraint that matches the input-output specification of the generator, we will try to apply this generator to find the value for the variable `x`. In practice, since there may be an infinite number of values satisfying a certain constraint, we will always put an upper bound on the number of values that we consider for each generator.

C Implementation Details for NAMO

Our most advanced strategy (interleaved-waypoint-connect) is implemented using the following CDL programs:

```
behavior NAMO(g: Pose):
goal: at(g)
body:
  bind x: Pose:
    valid_waypoint(x)
  bind t: Trajectory where:
    connect(x, g, t)
  promotable:
    achieve at(x)
    forall o: Object:
      achieve not blocking(o, t)
  do exec_traj(t)
eff:
  agent_position = g
```

```
15
16 # The base case of NAMO for the first waypoint.
17 behavior NAMO_base(g: Pose):
18 goal: at(g)
19 body:
20   bind t: Trajectory where:
21     connect(agent_position, g, t)
22   promotable:
23     forall o: Object:
24       achieve not blocking(o, t)
25   do exec_traj(t)
26 eff:
27   agent_position = g
28
29 behavior move_away(o: Object, t: Trajectory)
30 goal: not blocking(o, t)
31 body:
32   let original_pos = agent_position
33   # Move the agent to a position where it is close to o
34   achieve close_to(o)
35   do attach(o)
36   # Find a free location for the obstacle o away from t
37   bind o_pos = free_location(o, t, o_pos)
38   achieve position(o) == o_pos
39   do detach(o)
40   # Move back to the previous position
41   bind t: collision_free_connect(agent_position, original_pos, t)
42   do exec_traj(t)
43
44 behavior approach(o: Object):
45 goal: close_to(o)
46 body:
47   # Find a position for the agent that is close to the object
48   bind x: Pose where:
49     close_to_object(o, x)
50   # Find a trajectory from the agent's current position
51   #   to the target position
52   bind t: Trajectory where:
53     collision_free_connect(agent_position, x, t)
54   do exec_traj(t)
55 eff:
56   agent_position = x
57
58 behavior move_object_to(o: Object, p: Pose):
59 goal: position(o) == p
60 body:
61   # Find a trajectory for moving the object to the target position
62   bind t: Trajectory where:
63     collision_free_connect_with_object(position(o), p, t)
64   do exec_traj(t)
```

We implement the following basic generators for our NAMO domain.

1. `valid_waypoint`: we randomly sample a point that is not directly reachable from the initial state of the agent. Otherwise, it should be handled by the base case.
2. `connect`: we randomly sample a path (using A* search on the grid) that is not colliding with walls (i.e., non-movable obstacles). However, the path may collide with movable obstacles.
3. `free_location`: this is done by randomly sampling a position such that there is a collision-free path from the obstacle's current position to the sampled position, without considering the reachability of the robot. It is implemented by an A* search assuming the object itself can move freely.
4. `close_to_object`: this is done by randomly sampling a position that is within the reach of the robot but also being close to the target object.
5. `collision_free_connect`: this is implemented again, by using A* search on the grid for the robot moving from one location to another. It considers collisions with both movable and non-movable obstacles.
6. `collision_free_connect_with_object`: this function generates a trajectory of the object assuming that the object `o` has been attached to the object (so they will move jointly).

There are two features of these samplers that are worth noting. First, these generators all depend on the state of the world (i.e., the position of the agent and the position of all other objects). Second, we do not explicitly assert the existence of certain values. For example, when we are trying to move an obstacle to a new position, we do not assume that there is a path connecting the current robot position and the obstacle position. Such "assertions" happen implicitly during the CSP-solving process. Essentially, if we can not find a valid assignment to a variable (a pose `x` that is close to the object), the plan trace refinement fails.

Since all these generators execute non-deterministically, for a fair comparison between all different methods, we set the maximum number of intermediate waypoints that can be selected by all algorithms to 1.

D Manual of CDL

D.1 Key Concepts

Before we delve into the details of the programming language, we would like to start with a few key concepts in our design choices.

Object-centricness. In our language CDL, the state and the behavior representations are *lifted*. That is, we consider an environment composed of a set of entities. The state will be represented as features associated with entities or tuples of entities. Similarly, the behavior rules will optionally take entities as their

parameters and achieve certain entity properties. This would allow us to define a state and behavior representation that is *compact* and can encode a *family* of decision-making problems of indefinite numbers of objects, which is crucial for decision-making in physical environments.

As an example, in CDL, a state feature `position_2d` can be defined as

```
typedef Object
feature position_1d(x: Object) -> float32
```

In this case, the feature is unary, representing a single floating-point number associated with each entity in the environment. A behavior rule `move_to` can be defined as:

```
behavior move_to(x: Object, target: float32):
body:
  do control_move(x, target)
eff:
  position_1d[x] = target
```

The effect of the behavior rule is defined as updating the `position_1d` property of the entity `x`.

Data-oblivious computation graphs. The second important concept in CDL is *data oblivious computation*. In a nutshell, a *data-obvilious* program is a program whose:

1. data accesses do not depend on the input values;
2. all functions that combine data values are encapsulated into black-box operations;
3. furthermore, the control flow depends only on the input size (optionally random values to enable stochastic computation).

A signature example of non-oblivious computation is recursion. If our termination condition depends on the value of the input and does not have a simple upper bound of recursion depths, then we can not a priori construct the computation graph and apply it to any inputs. Actually, the decidability of the planning problem will also be undecidable (recall the halting problem, or Gödel's incompleteness theorem). Data-oblivious computation supports if-else conditions (which can be implemented by constructing both the `if` and the `else` branches), for-loops with constant bounds, or for-loops whose bounds only depends on the input size (the number of objects in the environment). It is a bit complex in construction but looping over all entities based on the value of a feature (smallest to biggest) can be implemented as a data-oblivious computation. But the compilation is inefficient, where the naïve implementation requires $O(N^2)$ circuit size. In practice, one may want to use other language features to realize this types of computation.

Data obliviousness will give us advantages in reasoning about policies or transition models written in the programming language. For example, if we consider

the policy as a program that generates a sequence of primitive controller commands, representing it as a data-oblivious program would allow us to derive the sequence of behavior commands without binding the parameters. For example, the following program is data-oblivious:

```
behavior move_five_steps(x: Object): # oblivious
  body:
    for i in range(5):
      do move_to(x, pose[x] -1)
```

In essence, this program outputs a fixed sequence of 5 `move_to` primitive controller calls. By contrast, the following program is not data-oblivious:

```
behavior move_until_less_than_0(x: Object): # non-oblivious
  body:
    while pose[x] > 0: # CDL does not support while
      do move_to(x, pose[x] -1)
```

Because we can't decide how many primitive controllers will be generated by this program without knowing the actual value of the initial `pose[x]`. Furthermore, from a program verification perspective, data-oblivious programs enable a simple way to reason about the precondition and the postcondition of a program, which we will explore in detail in later sections.

D.2 State Representation: Types and Features

For each decision-making problem, CDL assumes that there is a fixed and finite set of entities. This is not a very strong limitation. For entity creation, one can create additional "hypothetical" entities in the initial state whose properties will be filled after the entity is instantiated (although this assumes that we know an upper bound for the number of objects to be created). Similarly, for entity deletion, one can introduce a new unary feature `deleted` to record if an object has been deleted from the current state. Therefore, a world state, for any given problem, can be completely described in terms of values of a fixed set of basic *features* (of arity $0, 1, 2, \ldots$) of the given entities; the feature values may be continuous or discrete and may be high-dimensional.

In CDL, we make a distinction between *objects* (referring to entities in the physical world) and *values* (which are features or the return values of functions over these features). Both of them can be associated with types. An object type can be defined as:

```
  typedef T: BaseT
```

where `T` is the type name and `BaseT` is the name of the base type it inherits. When `BaseT` is omitted, it refines to the basic entity type in CDL. This can also be explicitly declared by inheriting the builtin type `object`. For example:

```
  typedef box
  typedef robot
```

These statements are equivalent to:

```
typedef box: object
typedef robot: object
```

And, a value type can be defined as:

```
typedef pose_1d: float32
typedef pose_2d: vector[float32; 2]
typedef pose_nd: vector[float32]
typedef python_class1: pyobject
```

Here, we are demonstrating a few different primitive types that a value type can inherit from, including a vector type of a fixed dimension, a vector type of an unknown dimension (whose dimension will be determined during program execution), and value types that correspond to Python types in the host program. As we will see in the next section, the key difference between objects and values lies in whether we can *quantify* over them. For example, we can say, for all objects x of a certain object type in the state, $f(x)$ is true. However, we can not state that, for all possible values v of a value type, $f(v)$ is true, because the set of all possible v's will be an infinite set in general.

Based on types, we have the feature definition:

```
feature f(v: T, ...)-> RT,
```

where `f` is the name of the feature, `v: T` declares an argument of name `v` and of type `T`. All features have a finite number of arguments. `RT` is the value type of the feature.

D.3 Functions

There are two types of functions in CDL: primitive functions and "derived" functions. Primitive functions are functions without a body. Their actual implementation resides in the external program (e.g., Python):

```
def f(v: T, ...)-> RT
```

This definition defines a function of name `f` and have an argument `v` of type `T`. It returns a value of type `RT`.

On the contrary, derived functions have a body and can compose the results of features and the return value of other functions (primitive or derived).

```
def f(v: T, ...)-> RT:
  return ...
```

The function bodies must be data-oblivious. In particular, we support the following features:

1. Read of features.

```
feature_1[arg_1, arg_2] # or
feature_1(arg_1, arg_2)
```

2. Logic operators, including finite-universe quantification operations.

```
f(arg_1) and f(arg_2)
f(arg_1) or f(arg_2)
not f(arg_1)
f(arg_1) ^ f(arg_2) # XOR
forall o: T: f(o)  # returns True if all entities of type T satisfies f
exists o: T: f(o)  # returns True if at least one entity
                   # of type T satisfies f
```

3. Basic arithmetic operations such as `+`, `-`, `*`, `/`, `**` (power).

```
f(arg_1) +f(arg_2)
f(arg_1) -f(arg_2)
f(arg_1) * f(arg_2)
f(arg_1) / f(arg_2)
f(arg_1) ** f(arg_2)
```

4. Assignments to local variables (`let`).
5. If-else conditions.

```
if c(...):
  statement_1
else:
  statement_2
```

6. "Parallel" for-loops over all entities that apply the same function over all entities. Assignments to local variables outside the scope of the for-loop is not allowed (`foreach`).

To ensure data-obliviousness, no self-recursion is supported. That is, it is not allowed to call a function `f` at any point of the execution of `f`. For example, the following definition of functions are invalid.

```
def f(x: float32) -> float32:
  return g(x)

def g(x: float32) -> float32:
  return f(x) # not allowed!
```

In this definition, even if we are not explicitly calling `f` itself inside `f`, we are implicitly creating self-recursion by calling `f` in a subroutine `g` that `f` invokes.

D.4 Non-Data-Oblivious Functions and Generators

In CDL, data-oblivious and non-oblivious functions can be mixed. Any data-oblivious computation can be illustrated as such a forward diagram, and in

CDL, such diagrams will always have a finite depth (but maybe "wide" at each depth due to quantifiers). CDL does not impose any constraints on the primitive functions (f, g, and h) in this case, therefore they can be generally non-data-oblivious and they will be implemented in external Python scripts. Another important reason to use such external primitive functions is that they allow us to write sophisticated programs to reason about numeric features and parameters, which are crucial for physical decision-making problems.

If our goal is to have the feature c be a particular value (e.g., at a particular location), we need to find "good" values for features a, b, c and d at the previous timestep. In CDL, since the computation graph is independent of the feature values, this problem can be modeled as a general constraint satisfaction problem.

If we only have a "forward" model of f, g, and h, there will be little we can do with this optimization process. A general technique to tackle such optimization problems would be to define additional inversion functions. Such functions are called "generators" in CDL. An example definition of a generator function is the following:

```
1 def f(x: float32, y: float32) -> float32
2
3 generator invert_f(x, y, z):
4   goal: f(x, y) == z
5   in: z
6   out: x, y
```

It declares an external function (e.g., `invert_f`) which takes as input the value of `z` and output two values `x` and `y` and it is guaranteed that `f(x, y) == z`.

D.5 Behavior Effects

Combining all these features, an effect section can use all operations supported in CDL, call to primitive and derived functions, to assign values to features. Since we have constrained the programs to be data-oblivious, an advantage (in addition to being able to formulate the constraint satisfaction problem) is that we can now translate such imperative transition models into a descriptive model, to possibly support other types of reasoning.

To be precise, there are two conventional paradigms for modeling T_a, the transition function associated with behavior a: imperative models (e.g., Python programs) and descriptive models (e.g., PDDL programs). In an imperative model, the effect of a is modeled as a function $T_a(s)$ that takes in the current state and generates a new state s'.

On the contrary, in a descriptive model, we write down a Hoare logic sentence with preconditions and effects. It has two formulas pre_a and eff_a and asserts:

$$\forall s. \{pre_a(s)\} \implies \{\forall (s' \sim T_a(s)) . eff_a(s')\}.$$

That is, both the pre_a and eff_a are descriptors of the current state and the next state. From a perspective of formal verification, the precondition and effects characterize the execution of the program T_a, and therefore transform the planning problem into a mathematical proving problem.

Both approaches have their pros and cons. The straightforward advantage of the imperative approach is that writing T_a by hand is usually very easy. In particular, if we allow the forward program T_a to be as flexible as Turing-computable functions, then writing the descriptive model of T_a by hand can be very challenging: intuitively, this would require the program writer to reason carefully about all effects related to the generation of a sound plan, very much like defining a semantic type system. The important advantage of the Hoare logic model is that now the solver can go both forward and backward. In particular, if our goal is to achieve a certain goal g, we can check the "effect" model of all actions to find relevant actions that can potentially achieve this goal.

The first empirical observation that we have about practical transition models (especially in physical decision-making problems) is that usually, we can constrain the forward model written in the imperative language (sacrificing the Turing completeness) but make it possible to derive Hoare logic statements from the forward model. The practice that CDL takes is that it restricts all effect sections to be data-oblivious. Furthermore, the corresponding circuit will always have a finite depth. CDL comes with a built-in compilation tool that can transform any function body into a holistic Haore logic description of their effects, which, subsequently, can be translated into a corresponding PDDL definition. For example, the following CDL program:

```
behavior do_something(x):
eff:
  for i in range(3):
    if f[x] > 0:
      f[x] -= 1
    else:
      break
```

can be translated into a precondition-effect modeling (in a PDDL-style language):

```
(:action
 :parameters (?x)
 :effect (and
   (= (f ?x)
     (condition (> (f ?x) 0)
       (condition (> (- (f ?x) 1) 0)
         (condition (> (- (- (f ?x) 1) 1) 0)
           (- (- (- (f ?x) 1) 1) 1)
           (- (- (f ?x) 1) 1)
         )
         (- (f ?x) 1)
       )
       (f ?x)
     )
```

```
    )
  )
)
```

D.6 Examples

The example below illustrates the ability to express a fine-grained ordering prior on subgoal achievement.

```
body:
  unordered:
    achieve open(cupboard)
    achieve holding(big_obj)
  let big_o_loc = loc_in(big_obj, cupboard)
  achieve loc[big_obj] == big_o_loc
  for each c: Cup:
    achieve contained_in(c, cupboard)
  achieve closed(cupboard)
```

The example below illustrates a CDL program for handling the classical STRIPS planning task of blocks world.

```
typedef block: object

# Features without return type annotations are assumed to be returning boolean values.

feature clear(x: block)
feature on(x: block, y: block)
feature on_table(x: block)
feature holding(x:block)
feature handempty()

controller pickup_table(x: block)
controller place_table(x: block)
controller stack(x: block, y: block)
controller unstack(x: block, y: block)

action r_holding_from_table(x: block):
  goal: holding(x)
  body:
    assert on_table(x)
    promotable sequential:
      achieve on_table(x)
      achieve clear(x)
    achieve handempty()
    pickup_table(x)
  eff:
    clear[x] = False
    handempty[...] = False
```

```
28      on_table[x] = False
29      holding[x] = True
30
31  action r_holding_from_stack(x: block):
32    goal: holding(x)
33    body:
34      bind y: block where:
35        on(x, y)
36      promotable sequential:
37        achieve on(x, y)
38        achieve clear(x)
39      achieve handempty()
40      unstack(x, y)
41    eff:
42      clear[x] = False
43      handempty[...] = False
44      on[x, y] = False
45      holding[y] = True
46      clear[y] = True
47
48  action r_clear(x: block):
49    goal: clear(x)
50    body:
51      bind y: block where:
52        on(y, x)
53      promotable sequential:
54        achieve on(y, x)
55        achieve clear(y)
56      achieve handempty()
57      unstack(y, x)
58    eff:
59      clear[y] = False
60      handempty[...] = False
61      on[y, x] = False
62      holding[y] = True
63      clear[x] = True
64
65  action r_clear_from_holding(x: block):
66    goal: clear(x)
67    body:
68      assert holding(x)
69      place_table(x)
70    eff:
71      holding[x] = False
72      clear[x] = True
73      handempty[...] = True
74      on_table[x] = True
75
76  action r_handempty():
77    goal: handempty()
```

```
  body:
    bind x: block where:
      holding(x)
    place_table(x)
  eff:
    holding[x] = False
    handempty[...] = True
    clear[x] = True
    on_table[x] = True

action r_on(x: block, y: block):
  goal: on(x, y)
  body:
    promotable sequential:
      achieve clear(y)
      achieve holding(x)
    stack(x, y)
  eff:
    clear[y] = False
    holding[x] = False
    on[x, y] = True
    clear[x] = True
    handempty[...] = True

action r_on_table(x: block):
  goal: on_table(x)
  body:
    promotable sequential:
      achieve holding(x)
    place_table(x)
  eff:
    holding[x] = False
    on_table[x] = True
    clear[x] = True
    handempty[...] = True
```

This definition would allow us to generate primitive plan solutions to planning problems such as the Sussman anomaly. Fig. 4 shows the refinement process from the high-level goal into a sound plan trace. If we need to describe the behavior using only sequential programs (but not partial orders), it would require a dedicated rule for the tuple of three objects. This figure also illustrates the refinement operator $\mathcal{R}^*$ in action. We will see in the next section that this will allow us to construct a full spectrum between purely imperative behavior programming and purely declarative planning-based behavior generation.

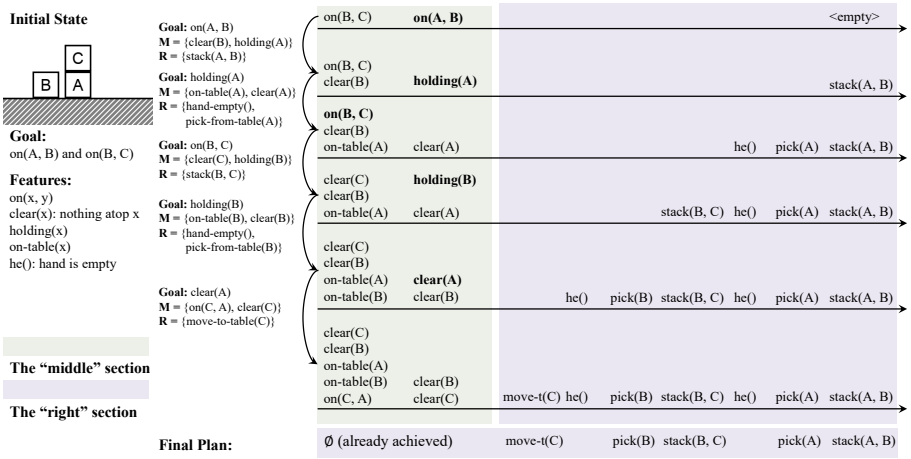


Fig. 4: Leveraging the promotable "middle" section of programs, we can plan for primitive plans where some of the preconditions have to be achieved in parallel with other preconditions or subgoals. The figure shows a plausible refinement trace for a primitive plan for the Sussman anomaly [26]. The refinement process starts with two subgoals $on(\mathrm{B},\mathrm{C})$ and $on(\mathrm{A},\mathrm{B})$, in this order. In the first step, it refines the subgoal $on(\mathrm{A},\mathrm{B})$ using the associated behavior rule. This gives us a new $M = \{on(\mathrm{A},\mathrm{B})), \{clear(\mathrm{B}), holding(\mathrm{A})\}_{seq}\}_{unordered}$ and a new $R = \{stack(\mathrm{A},\mathrm{B})\}_{seq}$. In the second step, it nondeterministically selected $holding(\mathrm{A})$ for refinement, and produces a new $M' = \{on(\mathrm{A},\mathrm{B})), clear(\mathrm{B}), \{on\text{-}table(\mathrm{A}), clear(\mathrm{A})\}_{seq}\}_{unordered}$ and a new $R' = \{hand\text{-}empty(), pick\text{-}from\text{-}table(\mathrm{A}), stack(\mathrm{A},\mathrm{B})\}_{seq}$. This refinement process continues until we have a fully refined and sound plan.